DSP56800E_Quick_Start User's Manual

Targeting Freescale 56F8xxx Platform

Rev. 2.4, 01/04/2009

freescale semiconductor

This document contains information on a new product. Specifications and information herein are subject to change without notice.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All opening parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale was negligent regarding the design or manufacture of the part. Freescale is registered trademarks of Freescale Semiconductor, Inc. Freescale Semiconductor, Inc. is an Equal Opportunity/Affirmative Action Employer. All other tradenames, trademarks, and registered trademarks are the property of their respective owners.

How to reach us:

North and South America +1 800 521 6274 6AM-5PM Mountain Standard Time

Europe: 9AM-5PM Central European Time

English +44 1296 380 456 or +46 8 52200080

German +49 89 92103 559

French +33 1 69 35 48 48

Asia:

Asia Pacific Region excl.India +800 2666 8080 8AM-6PM Hong Kong

India 000 800 852 1155 8AM-6PM Hong Kong

Japan 0120 191 014 8AM-5PM Tokyo

HOME PAGE: http://www.freescale.com/

© Copyright Freescale, Inc., 2009

Chapter 1
Introduction

1.1 Overview 1-1
1.1.1 Features 1-2
1.1.1.1 Core-system Infrastructure 1-2
1.1.1.2 On-chip Drivers 1-2
1.1.1.3 Sample Applications 1-2
1.1.1.4 Graphical Configuration Tool 1-3
1.1.1.5 FreeMASTER Software 1-3
1.2 Quick Start 1-4
1.2.1 Install CodeWarrior Development Tools 1-4
1.2.2 Install DSP56800E_Quick_Start 1-5
1.2.2.1 Supplementary DSP56800E_Quick_Start Installation Steps 1-6
1.2.2.2 Install Graphical Configuration Tool 1-6
1.2.2.3 Install FreeMASTER (PC Master Software) 1-8
1.2.3 Install MC56F8xxxEVM Hardware 1-8
1.2.4 Build and Run Sample Application 1-10

Chapter 2
Core System Infrastructure

2.1 Boot Sequence 2-1
2.1.1 Power-up/Reset 2-2
2.1.2 Start() - entry point 2-3
2.1.3 userPreMain() 2-3
2.1.4 main() the User's Application Code 2-3
2.1.5 userPostMain() 2-4
2.2 Data Types 2-5
2.3 ArchIO Peripheral Register Structures 2-6
2.4 Core System's Routines and Macros 2-7
2.4.1 Architecture dependent routines 2-7
2.4.1.1 archEnableInt - enable interrupts 2-7
2.4.1.2 archEnableIntLvl123 - enable interrupt levels 1, 2 and 3 2-7
2.4.1.3 archEnableIntLvl23 - enable interrupts levels 2 and 3 2-8
2.4.1.4 archDisableInt - disable interrupts 2-8
2.4.1.5 archResetLimitBit - reset limit bit 2-8
2.4.1.6 archSetNoSat - set no saturation mode 2-9
2.4.1.7 archSetSat32 - set saturation mode 2-9
2.4.1.8 archSet2CompRound - set two's complement rounding mode 2-9
2.4.1.9 archSetConvRound - set convergent rounding mode 2-10

2.4.1.10 archStop - stop processing state 2-10
2.4.1.11 archTrap - initiate a software interrupt 2-10
2.4.1.12 archWait - wait processing state 2-11
2.4.1.13 archGetLimitBit - get limit bit 2-11
2.4.1.14 archGetSetSaturationMode - get and set saturation mode 2-11
2.4.1.15 archDelay - delay 2-12
2.4.2 Macros for peripheral memory access 2-13
2.4.2.1 periphMemRead - memory read 2-13
2.4.2.2 periphMemWrite - memory write 2-14
2.4.2.3 periphBitSet - set selected bits 2-14
2.4.2.4 periphMemInvBitSet - invert memory content and set selected bits 2-15
2.4.2.5 periphBitClear - clear selected bits 2-15
2.4.2.6 periphBitGrpSR - set bit group to given value 2-16
2.4.2.7 periphBitGrpRS - set bit group to given value 2-16
2.4.2.8 periphBitGrpZS - set bit group to given value 2-17
2.4.2.9 periphBitGrpSet - set bit group to given value 2-18
2.4.2.10 periphSafeAckByOne - clear (acknowledge) bit flags which are active-high and are cleared by write-one2-19
2.4.2.11 periphBitChange - change selected bits 2-19
2.4.2.12 periphBitTest - test selected bits 2-20
2.4.3 Miscellaneous Routines 2-21
2.4.3.1 impyuu - integer multiply unsigned 16b x unsigned 16b 2-21
2.4.3.2 impysu - integer multiply signed 16b x unsigned 16b 2-21
2.4.3.3 shl2 - optimized version of shl intrinsic function 2-22
2.4.3.4 shr2 - optimized version of shr intrinsic function 2-23
2.4.4 Intrinsic Functions 2-23
2.5 Interrupts 2-24
2.5.1 Processing Interrupts 2-24
2.5.1.1 Interrupt Vector Table 2-24
2.5.1.2 Interrupt Processing Flow 2-24
2.5.1.3 ISRs 2-25
2.5.1.4 Interrupt Priority Levels 2-26
2.5.1.5 Fast Interrupts 2-26
2.5.1.6 Clearing Interrupt Flags 2-26
2.5.2 Configuring Interrupts 2-27
2.5.2.1 Installing ISRs 2-27
2.5.2.2 Assigning Interrupt Priority Levels 2-27
2.5.2.3 Installing Fast Interrupts 2-28
2.5.2.4 Enabling Interrupts 2-29
2.5.3 Code Example 2-29
2.6 Advanced Topics 2-34
2.6.1 Project Targets 2-34
2.6.2 Inside Startup Code 2-39
2.6.2.1 Symbols Used in Startup Code 2-39
2.6.2.2 Startup Source Code 2-41

Chapter 3

Directory Structure

3.1 Root Directory 3-1
3.2 Sample Applications Directory 3-2
3.3 Tools Directory 3-2
3.4 Src Directory 3-3
3.5 Stationery Directory 3-3
3.6 User_manuals Directory 3-4

Chapter 4
Developing Software

4.1 Creating a new project 4-1
4.2 On-chip peripheral initialization 4-2
4.3 On-chip drivers - interface description 4-3
4.3.1 ioctl() 4-4
4.3.2 read() 4-5
4.3.3 write() 4-5
4.4 Interrupts and Interrupt Service Routines 4-6
4.5 appconfig.h file 4-6

Chapter 5
On-chip Drivers

5.1 OCCS Driver 5-5
5.1.1 Introduction 5-5
5.1.2 Quick Reference 5-5
5.1.2.1 OCCS frequency calculation 5-5
5.1.2.2 API Definition 5-6
5.1.2.3 Configuration Items 5-6
5.1.2.4 API Specification 5-7
5.1.3 Detailed API Specification 5-12
5.1.4 OCCS Driver Sample Application 5-53
5.2 INTC Driver 5-61
5.2.1 Introduction 5-61
5.2.2 Quick Reference 5-61
5.2.2.1 API Definition 5-61
5.2.2.2 Configuration Items 5-62
5.2.2.3 API Specification 5-63
5.2.3 Detailed API Specification 5-65
5.2.4 INTC Driver Application 5-81
5.3 WINTC Driver 5-87
5.3.1 Introduction 5-87
5.3.2 Quick Reference 5-87
5.3.2.1 API Definition 5-87
5.3.2.2 Configuration Items 5-88
5.3.2.3 API Specification 5-89
5.3.3 Detailed API Specification 5-91

5.3.4 INTC Driver Application 5-103
5.4 COP Driver 5-111
5.4.1 Introduction. 5-111
5.4.2 Quick Reference 5-111
5.4.2.1 API Definition. 5-111
5.4.2.2 Configuration Items 5-112
5.4.2.3 API Specification 5-112
5.4.3 Detailed API Specification 5-114
5.4.4 COP Driver Application 5-128
5.5 SYS Driver 5-133
5.5.1 Introduction. 5-133
5.5.2 Quick Reference 5-133
5.5.2.1 API Definition. 5-134
5.5.2.2 Configuration Items 5-134
5.5.2.3 API Specification 5-136
5.5.3 Detailed API Specification 5-144
5.5.4 SYS Driver Application 5-202
5.6 PMC Driver. 5-209
5.6.1 Introduction. 5-209
5.6.2 Quick Reference 5-209
5.6.2.1 API Definition. 5-209
5.6.2.2 Configuration Items 5-210
5.6.2.3 API Specification 5-210
5.6.3 Detailed API Specification 5-212
5.6.4 PMC Driver Applications 5-233
5.6.4.1 pmc_demo. 5-233
5.7 FlexCAN Driver 5-235
5.7.1 Introduction. 5-235
5.7.2 Quick Reference 5-236
5.7.2.1 FlexCAN Bit-Timing 5-236
5.7.2.2 FlexCAN Message Identifiers 5-237
5.7.2.3 API Definition. 5-239
5.7.2.4 Configuration Items 5-239
5.7.2.5 API Specification 5-240
5.7.3 Detailed API Specification 5-246
5.7.4 FlexCAN Driver Sample Application. 5-299
5.8 GPIO Driver 5-307
5.8.1 Introduction. 5-307
5.8.2 Quick Reference 5-307
5.8.2.1 API Definition. 5-307
5.8.2.2 Configuration Items 5-308
5.8.2.3 API Specification 5-309
5.8.3 Detailed API Specification 5-311
5.8.4 GPIO Driver Application 5-344
5.9 ADC Driver (MC56F83xx,MC56F801x,MC56F802x/3x) 5-349
5.9.1 Introduction. 5-349
5.9.2 Quick Reference 5-349

5.9.2.1 API Definition. 5-349
5.9.2.2 Configuration Items 5-350
5.9.2.3 API Specification 5-352
5.9.3 Detailed API Specification 5-359
5.9.4 ADC Driver Application. 5-428
5.10 ADC Driver (MC56F800x) 5-433
5.10.1 Introduction. 5-433
5.10.2 Quick Reference 5-433
5.10.2.1 API Definition. 5-433
5.10.2.2 Configuration Items 5-434
5.10.2.3 API Specification 5-434
5.10.3 Detailed API Specification 5-437
5.10.4 ADC Driver Applications 5-455
5.10.4.1 adc_demo 5-455
5.11 PGA Driver 5-457
5.11.1 Introduction. 5-457
5.11.2 Quick Reference 5-457
5.11.2.1 API Definition. 5-457
5.11.2.2 Configuration Items 5-458
5.11.2.3 API Specification 5-458
5.11.3 Detailed API Specification 5-460
5.11.4 PGA Driver Applications 5-478
5.11.4.1 pga_demo 5-478
5.12 PDB Driver 5-479
5.12.1 Introduction. 5-479
5.12.2 Quick Reference 5-479
5.12.2.1 API Definition. 5-479
5.12.2.2 Configuration Items 5-480
5.12.2.3 API Specification 5-480
5.12.3 Detailed API Specification 5-482
5.12.4 PDB Driver Applications 5-498
5.12.4.1 PDB_demo 5-498
5.13 Quadrature Decoder Driver. 5-499
5.13.1 Introduction. 5-499
5.13.2 Quick Reference 5-499
5.13.2.1 API Definition. 5-499
5.13.2.2 Configuration Items 5-502
5.13.2.3 API Specification 5-502
5.13.3 Detailed API Specification 5-506
5.13.4 Quadrature Decoder Driver Application. 5-542
5.14 PWM Driver 5-547
5.14.1 Introduction. 5-547
5.14.2 Quick Reference 5-547
5.14.2.1 PWM Module Setting 5-547
5.14.2.2 API Definition. 5-548
5.14.2.3 Configuration Items 5-550
5.14.2.4 API Specification 5-552

5.14.3 Detailed API Specification 5-560
5.14.4 PWM Driver Application 5-637
5.15 SCI Driver 5-645
5.15.1 Introduction 5-645
5.15.2 Quick Reference 5-645
5.15.2.1 API Definition 5-645
5.15.2.2 Configuration Items 5-646
5.15.2.3 API Specification 5-648
5.15.3 Detailed API Specification 5-653
5.15.4 SCI Driver Application 5-706
5.16 SPI Driver 5-713
5.16.1 Introduction 5-713
5.16.2 Quick Reference 5-713
5.16.2.1 API Definition 5-713
5.16.2.2 Configuration Items 5-714
5.16.2.3 API Specification 5-715
5.16.3 Detailed API Specification 5-720
5.16.4 SPI Driver Application 5-765
5.17 IIC Driver (MC56F801x,MC56F800x) 5-773
5.17.1 Introduction 5-773
5.17.1.1 IIC Bus Transactions 5-773
5.17.2 Quick Reference 5-774
5.17.2.1 API Definition 5-775
5.17.2.2 Configuration Items 5-775
5.17.2.3 API Specification 5-777
5.17.3 Detailed API Specification 5-780
5.17.4 IIC Driver Sample Application 5-817
5.18 IIC Driver (MC56F802x/3x) 5-819
5.18.1 Introduction 5-819
5.18.2 Quick Reference 5-819
5.18.2.1 API Definition 5-820
5.18.2.2 Configuration Items 5-820
5.18.2.3 API Specification 5-821
5.18.3 Detailed API Specification 5-825
5.18.4 IIC Driver Applications 5-859
5.19 Temperature Sensor System Driver 5-861
5.19.1 Introduction 5-861
5.19.2 Quick Reference 5-861
5.19.2.1 API Definition 5-861
5.19.2.2 Configuration Items 5-862
5.19.2.3 API Specification 5-862
5.19.3 Detailed API Specification 5-863
5.19.4 Temperature Sensor System Driver Application 5-867
5.20 Quad Timer Driver 5-869
5.20.1 Introduction 5-869
5.20.2 Quick Reference 5-869
5.20.2.1 API Definition 5-869

5.20.2.2 Configuration Items 5-870
5.20.3 Detailed API Specification 5-875
5.20.4 Quad Timer Driver Application 5-924
5.21 PIT Driver 5-931
5.21.1 Introduction 5-931
5.21.2 Quick Reference 5-931
5.21.2.1 API Definition 5-931
5.21.2.2 Configuration Items 5-932
5.21.2.3 API Specification 5-932
5.21.3 Detailed API Specification 5-934
5.21.4 PIT Driver Applications 5-944
5.21.4.1 dac_pit_demo 5-944
5.21.4.2 dac_cmp_demo 5-944
5.21.4.3 adc_fmstr_demo 5-944
5.22 CMP Driver 5-945
5.22.1 Introduction 5-945
5.22.2 Quick Reference 5-945
5.22.2.1 API Definition 5-945
5.22.2.2 Configuration Items 5-946
5.22.2.3 API Specification 5-946
5.22.3 Detailed API Specification 5-948
5.22.4 CMP Driver Applications 5-962
5.22.4.1 dac_cmp_demo 5-962
5.23 HSCMP Driver 5-963
5.23.1 Introduction 5-963
5.23.2 Quick Reference 5-963
5.23.2.1 API Definition 5-963
5.23.2.2 Configuration Items 5-964
5.23.2.3 API Specification 5-964
5.23.3 Detailed API Specification 5-966
5.23.4 HSCMP Driver Applications 5-986
5.23.4.1 hscmp_demo 5-986
5.24 DAC Driver 5-987
5.24.1 Introduction 5-987
5.24.2 Quick Reference 5-987
5.24.2.1 API Definition 5-987
5.24.2.2 Configuration Items 5-988
5.24.2.3 API Specification 5-988
5.24.3 Detailed API Specification 5-990
5.24.4 DAC Driver Applications 5-1008
5.24.4.1 dac_pit_demo 5-1008
5.24.4.2 dac_cmp_demo 5-1008
5.24.4.3 adc_fmstr_demo 5-1008
5.25 MSCAN Driver 5-1009
5.25.1 Introduction 5-1009
5.25.2 Quick Reference 5-1009
5.25.2.1 MSCAN Bit-Timing 5-1009

5.25.2.2 CAN Message Identifiers 5-1011
5.25.2.4 Configuration Items 5-1013
5.25.2.5 API Specification 5-1014
5.25.3 Detailed API Specification 5-1020
5.25.4 Message Buffer API Specification 5-1074
5.25.5 MSCAN Driver Application 5-1088
5.26 RTC Driver 5-1089
5.26.1 Introduction 5-1089
5.26.2 Quick Reference 5-1089
5.26.2.1 API Definition 5-1089
5.26.2.2 Configuration Items 5-1090
5.26.2.3 API Specification 5-1090
5.26.3 Detailed API Specification 5-1092
5.26.4 RTC Driver Applications 5-1104
5.26.4.1 rtc_demo 5-1104

Chapter 6
FreeMASTER Driver

6.1 Introduction 6-1
6.2 Driver Files 6-2
6.3 Interrupt Handling 6-3
6.4 New Features 6-4
6.4.1 Target-side Addressing 6-4
6.4.2 Application Command Callbacks 6-4
6.5 Driver Configuration 6-4
6.6 Driver Usage 6-10
6.6.1 Driver API 6-10
6.6.2 Code Listing: freemaster_demo 6-24
6.6.3 Code Listing: freemaster_demo2 6-27

Chapter 7
Graphical Configuration Tool

7.1 Introduction 7-1
7.1.1 Features 7-1
7.1.2 How does it work? 7-2
7.2 Program usage 7-3
7.2.1 GUI Description 7-3
7.2.1.1 Peripheral Modules Tree 7-4
7.2.1.2 Peripheral Module Settings Pane 7-4
7.2.1.3 Pinout Page 7-4
7.2.1.4 Register View 7-5
7.2.1.5 Warnings View 7-6
7.2.1.6 Options dialog 7-6
7.2.2 Application Configuration File Structure 7-7

Chapter 8

License

8.1 Software License Agreement 8-1

List of Tables

2-1 archGetSetSaturationMode arguments 2-11
2-2 archDelay arguments 2-12
2-3 periphMemRead arguments 2-13
2-4 periphMemWrite arguments 2-14
2-5 periphBitSet arguments. 2-14
2-6 periphMemInvBitSet arguments. 2-15
2-7 periphBitClear arguments 2-15
2-8 periphBitSet arguments. 2-16
2-9 periphBitSet arguments. 2-16
2-10 periphBitSet arguments. 2-17
2-11 periphBitSet arguments. 2-18
2-12 periphSafeAckByOne arguments 2-19
2-13 periphBitChange arguments 2-19
2-14 periphBitTest arguments. 2-20
2-15 impyuu arguments. 2-21
2-16 impysu arguments 2-21
2-17 shl2 arguments 2-22
2-18 shr2 arguments 2-23
2-19 Targets of the MC56F8300DEMO project. 2-34
2-20 Targets of the MC56F8323EVM project.. 2-34
2-21 Targets of the MC56F8346EVM project.. 2-34
2-22 Targets of the MC56F8346CB project. 2-35
2-23 Targets of the MC56F8357EVM project.. 2-36
2-24 Targets of the MC56F8367EVM project.. 2-36
2-25 Targets of the MC56F8013DEMO and MC56F8014DEMO project. 2-37
2-26 Targets of the MC56F8013CB project. 2-37
2-27 Targets of the MC56F8023DEMO, MC56F8023CB and MC56F8025DEMO project.2-37
2-28 Targets of the MC56F8037EVM project.. 2-38
5-1 OCCS Module Base Address 5-5
5-2 OCCS Configuration Items for appconfig.h 5-6
5-3 OCCS Driver Arguments - ioctl 5-8
5-4 ioctl commands 5-8
5-5 OCCS_INIT ioctl call arguments 5-13

5-6 OCCS_SET_CORE_CLOCK ioctl call arguments 5-14
5-7 OCCS_SET_POSTSCALER ioctl call arguments 5-15
5-8 OCCS_SET_PRESCALER ioctl call arguments 5-16
5-9 OCCS_SET_DIVIDE_BY ioctl call arguments 5-17
5-10 OCCS_INT_ENABLE ioctl call arguments 5-18
5-11 OCCS_INT_DISABLE ioctl call arguments 5-19
5-12 OCCS_LOCK_DETECTOR ioctl call arguments 5-20
5-13 OCCS_TURN_OFF_CHARGE_PUMP ioctl call arguments 5-21
5-14 OCCS_SET_ZCLOCK SOURCE ioctl call arguments 5-22
5-15 OCCS_GET_ZCLOCK SOURCE ioctl call arguments 5-23
5-16 OCCS_READ_FLAG ioctl call arguments 5-24
5-17 OCCS_CLEAR_FLAG ioctl call arguments 5-26
5-18 OCCS_GET_IPBUS_FREQ ioctl call arguments 5-27
5-19 OCCS_SET_LORTP ioctl call arguments 5-28
5-20 OCCS_WRITE_CONTROL_REG ioctl call arguments 5-29
5-21 OCCS_WRITE_DIVIDE_BY_REG ioctl call arguments 5-30
5-22 OCCS_WRITE_OSC_CONTROL_REG ioctl call arguments 5-31
5-23 OCCS_READ_CONTROL_REG ioctl call arguments 5-32
5-24 OCCS_READ_DIVIDE_BY_REG ioctl call arguments 5-33
5-25 OCCS_READ_STATUS_REG ioctl call arguments 5-34
5-26 OCCS_READ_OSC_CONTROL_REG ioctl call arguments 5-35
5-27 OCCS_POWER_MODE ioctl call arguments 5-36
5-28 OCCS_SHUTDOWN ioctl call arguments. 5-37
5-29 OCCS_SET_ZCLOCK OCCS_SET_PRESCALER_CLOCK ioctl call arguments 5-38
5-30 OCCS_INTERNAL_RELAX_OSC_OPERATION ioctl call arguments 5-39
5-31 OCCS_ADJUST_RELAX_OSC_FREQ ioctl call arguments 5-40
5-32 OCCS_TRIM_RELAX_OSC_8MHZ ioctl call arguments 5-41
5-33 OCCS_DIRECT_CLOCK_MODE ioctl call arguments 5-42
5-34 OCCS_SELECT_EXT_CLOCK_SOURCE ioctl call arguments 5-43
5-35 OCCS_WPROTECT_PLL_SETTINGS ioctl call arguments 5-45
5-36 OCCS_WPROTECT_OSC_SETTINGS ioctl call arguments 5-46
5-37 OCCS_WPROTECT_CLK_SETTINGS ioctl call arguments 5-47
5-38 OCCS_SET_CLOCK_CHECK ioctl call arguments 5-48
5-39 OCCS_TEST_CLOCK_CHECK ioctl call arguments. 5-49
5-40 OCCS_READ_CLOCK_CHECK_REFERENCE ioctl call arguments. 5-50
5-41 OCCS_READ_CLOCK_CHECK_TARGET ioctl call arguments 5-51
5-42 OCCS_SELECT_FREQ_RANGE ioctl call arguments. 5-52
5-43 INTC Module Base Address. 5-61

5-44 INTC Configuration Items for appconfig.h 5-62
5-45 INTC Driver Arguments - ioctl 5-63
5-46 ioctl commands 5-63
5-47 INTC_INIT ioctl call arguments. 5-66
5-48 INTC_INTERRUPTS ioctl call arguments 5-67
5-49 INTC_SET_IPL_n ioctl call arguments 5-68
5-50 INTC_SET_IPL_n_RAW ioctl call arguments 5-69
5-51 INTC_GET_IPL_n_RAW ioctl call arguments 5-70
5-52 INTC_SET_FASTINTx ioctl call arguments 5-71
5-53 INTC_SET_FASTINTx_VEC ioctl call arguments 5-72
5-54 INTC_GET_PENDING_FLAG ioctl call arguments 5-73
5-55 INTC_READ_CONTROL_REG ioctl call arguments 5-74
5-56 INTC_GET_INT_STATE ioctl call arguments 5-75
5-57 INTC_GET_INT_LEVEL ioctl call arguments 5-76
5-58 INTC_GET_INT_NUMBER ioctl call arguments 5-77
5-59 INTC_READ_IRQPINS ioctl call arguments 5-78
5-60 INTC_SELECT_EDGE_MODE ioctl call arguments 5-79
5-61 INTC_SELECT_LEVEL_MODE ioctl call arguments 5-80
5-62 WINTC Module Base Address 5-87
5-63 WINTC Configuration Items for appconfig.h 5-88
5-64 WINTC Driver Arguments - ioctl. 5-89
5-65 ioctl commands 5-89
5-66 WINTC_INIT ioctl call arguments. 5-92
5-67 WINTC_INTERRUPTS ioctl call arguments. 5-93
5-68 WINTC_GET_INT_STATE ioctl call arguments 5-94
5-69 WINTC_GET_INT_LEVEL ioctl call arguments 5-95
5-70 WINTC_GET_INT_NUMBER ioctl call arguments 5-96
5-71 WINTC_ASSIGN_USERx_VECTOR ioctl call arguments 5-97
5-72 WINTC_SET_INT_EOnCE_STEP_COUNTER ioctl call arguments 5-98
5-73 WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL ioctl call arguments 5-99
5-74 WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY ioctl call arguments5-100
5-75 WINTC_SET_INT_EOnCE_TRACE_BUFFER ioctl call arguments 5-101
5-76 WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0 ioctl call arguments . 5-102
5-77 COP Module Base Address 5-111
5-78 COP Configuration Items for appconfig.h 5-112
5-79 COP Driver Arguments - ioctl 5-112
5-80 ioctl commands 5-113

5-81 COP_INIT ioctl call arguments 5-115
5-82 COP_DEVICE ioctl call arguments 5-116
5-83 COP_SET_TIMEOUT ioctl call arguments 5-117
5-84 COP_READ_COUNTER ioctl call arguments 5-118
5-85 COP_CLEAR_COUNTER ioctl call arguments 5-119
5-86 COP_CLEAR_COUNTER_PART1 ioctl call arguments 5-120
5-87 COP_CLEAR_COUNTER_PART2 ioctl call arguments 5-121
5-88 COP_RUN_IN_STOP ioctl call arguments 5-122
5-89 COP_RUN_IN_WAIT ioctl call arguments 5-123
5-90 COP_WRITE_PROTECT ioctl call arguments 5-124
5-91 COP_LOR_WATCHDOG ioctl call arguments 5-125
5-92 COP_SET_CLOCK_SOURCE ioctl call arguments 5-126
5-93 COP_SET_CLOCK_PRESCALER ioctl call arguments 5-127
5-94 Module Base Address 5-133
5-95 SYS Configuration Items for appconfig.h MC56F83xx/MC56F801x 5-134
5-96 SYS Configuration Items for appconfig.h MC56F802x/3x 5-135
5-97 SYS Configuration Items for appconfig.h MC56F800x 5-135
5-98 SYS Driver Arguments - ioctl 5-136
5-99 ioctl commands 5-136
5-100 SYS_INIT ioctl call arguments 5-145
5-101 SYS_STOP ioctl call arguments 5-146
5-102 SYS_STOP_PERMANENT_DISABLE ioctl call arguments 5-147
5-103 SYS_WAIT ioctl call arguments 5-148
5-104 SYS_WAIT_PERMANENT_DISABLE ioctl call arguments 5-149
5-105 SYS_SOFTWARE_RESET ioctl call arguments 5-150
5-106 SYS_ONCE ioctl call arguments 5-151
5-107 SYS_WRITE_SW_CONTROL_REGn ioctl call arguments 5-152
5-108 SYS_READ_SW_CONTROL_REGn ioctl call arguments 5-153
5-109 SYS_READ_LSH_JTAG_ID ioctl call arguments 5-154
5-110 SYS_READ_MSH_JTAG_ID ioctl call arguments 5-155
5-111 SYS_TEST_RESET_SOURCE ioctl call arguments 5-156
5-112 SYS_CLEAR_RESET_SOURCE ioctl call arguments 5-158
5-113 SYS_PULL_UP_ENABLE ioctl call arguments 5-160
5-114 SYS_PULL_UP_DISABLE ioctl call arguments 5-161
5-115 SYS_CLKOUT ioctl call arguments 5-162
5-116 SYS_CLKOUT_SELECT ioctl call arguments 5-163
5-117 SYS_CLKOUT_ENABLE ioctl call arguments 5-164
5-118 SYS_CLKOUT_DISABLE ioctl call arguments 5-165
5-119 SYS_SET_CLKOUT_0_SOURCE ioctl call arguments 5-166

5-120 SYS_SET_CLKOUT_1_SOURCE ioctl call arguments 5-167
5-121 SYS_ACLK_ENABLE ioctl call arguments 5-168
5-122 SYS_ACLK_DISABLE ioctl call arguments. 5-169
5-123 SYS_SET_PADS_FUNCTION ioctl call arguments. 5-170
5-124 SYS_ENABLE_INTERNAL_TMR_SIGNAL ioctl call arguments 5-171
5-125 SYS_DISABLE_INTERNAL_TMR_SIGNAL ioctl call arguments 5-172
5-126 SYS_PERIPH_CLK_ENABLE ioctl call arguments. 5-173
5-127 SYS_PERIPH_CLK_DISABLE ioctl call arguments 5-174
5-128 SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG ioctl call arguments 5-175
5-129 SYS_READ_IO_SHORT_ADDR_LOCATION_REG ioctl call arguments. 5-176
5-130 SYS_ENABLE_IN_STOP ioctl call arguments. 5-177
5-131 SYS_DISABLE_IN_STOP ioctl call arguments 5-178
5-132 SYS_SET_isig_SOURCE ioctl call arguments 5-179
5-133 SYS_SET_pad_FUNCTION ioctl call arguments 5-182
5-134 SYS_HS_CLOCK_ENABLE ioctl call arguments 5-183
5-135 SYS_HS_CLOCK_DISABLE ioctl call arguments. 5-184
5-136 SYS_SET_POWER_MODE ioctl call arguments 5-185
5-137 SYS_GET_POWER_MODE ioctl call arguments. 5-186
5-138 SYS_WPROTECT_CLOCK_SETTINGS ioctl call arguments. 5-187
5-139 SYS_WPROTECT_SIGNALS_ROUTING ioctl call arguments 5-188
5-140 LVI_GET_LOW_VOLTAGE ioctl call arguments 5-189
5-141 LVI_GET_NONSTICKY_INT_SOURCE ioctl call arguments 5-190
5-142 LVI_CLEAR_LOW_VOLTAGE_INT ioctl call arguments 5-191
5-143 LVI_INT_ENABLE ioctl call arguments. 5-192
5-144 LVI_INT_DISABLE ioctl call arguments 5-193
5-145 LVI_INT_SELECT ioctl call arguments 5-194
5-146 SEMI_SET_DRIVE_BUS ioctl call arguments 5-195
5-147 SEMI_WRITE_BASEREGn ioctl call arguments 5-196
5-148 SEMI_WRITE_OPTIONREGn ioctl call arguments. 5-197
5-149 SEMI_WRITE_CONTROLREG ioctl call arguments. 5-198
5-150 SEMI_READ_BASEREGn ioctl call arguments 5-199
5-151 SEMI_READ_OPTIONREGn ioctl call arguments. 5-200
5-152 SEMI_READ_CONTROLREG ioctl call arguments. 5-201
5-153 PMC Module Base Address 5-209
5-154 PMC Configuration Items for appconfig.h. 5-210
5-155 PMC Driver Arguments - ioctl 5-210
5-156 ioctl commands. 5-211
5-157 PMC_INIT ioctl call arguments 5-213
5-158 PMC_CLEAR_FLAGS ioctl call arguments 5-214

5-159 PMC_TEST_FLAGS ioctl call arguments 5-215
5-160 PMC_SET_INT_ENABLE ioctl call arguments 5-216
5-161 PMC_SET_INT_DISABLE ioctl call arguments. 5-217
5-162 PMC_SET_LOW_VOLTAGE_RESET ioctl call arguments 5-218
5-163 PMC_SET_PARTIAL_POWER_DOWN ioctl call arguments. 5-219
5-164 PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES ioctl call arguments 5-220
5-165 PMC_TEST_LOW_POWER_REGULATOR_STATUS ioctl call arguments 5-221
5-166 PMC_SET_LOW_POWER_WAKEUP_INTERRUPT ioctl call arguments 5-222
5-167 PMC_SET_BANDGAP_BUFFER ioctl call arguments 5-223
5-168 PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE ioctl call arguments 5-224
5-169 PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE ioctl call arguments. . . . 5-225
5-170 PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL ioctl call arguments. 5-226
5-171 PMC_SET_WPROTECTION ioctl call arguments 5-227
5-172 PMC_SET_1KHZ_OSC ioctl call arguments 5-228
5-173 PMC_SET_1KHZ_OSC_TRIM ioctl call arguments 5-229
5-174 PMC_SET_1KHZ_OSC_FACTORY_TRIM ioctl call arguments 5-230
5-175 PMC_SET_LVD_TRIM ioctl call arguments 5-231
5-176 PMC_SET_LVD_FACTORY_TRIM ioctl call arguments 5-232
5-177 FlexCAN Module Base Address. 5-236
5-178 FlexCAN Configuration Items for appconfig.h 5-239
5-179 FlexCAN Driver Arguments - ioctl 5-240
5-180 FlexCAN Module ioctl Commands 5-241
5-181 FlexCAN Module ioctl Commands 5-245
5-182 FCAN_INIT ioctl call arguments 5-247
5-183 FCAN_STOP_MODE ioctl call arguments 5-248
5-184 FCAN_DEBUG_MODE ioctl call arguments 5-249
5-185 FCAN_SOFT_RESET ioctl call arguments 5-250
5-186 FCAN_SELF_WAKEUP_MODE ioctl call arguments. 5-251
5-187 FCAN_AUTO_PWRSAVE_MODE ioctl call arguments 5-252
5-188 FCAN_TEST_READY ioctl call arguments 5-253
5-189 FCAN_TEST_DEBUG ioctl call arguments 5-254
5-190 FCAN_TEST_STOP ioctl call arguments 5-255
5-191 FCAN_INT_ENABLE ioctl call arguments. 5-256
5-192 FCAN_INT_DISABLE ioctl call arguments 5-257
5-193 FCAN_LOOPBACK_MODE ioctl call arguments 5-258
5-194 FCAN_TIMER_SYNC_MODE ioctl call arguments 5-259

5-195 FCAN_LISTEN_ONLY_MODE ioctl call arguments. 5-260
5-196 FCAN_SET_TX_FIRST_SCHEME ioctl call arguments 5-261
5-197 FCAN_SET_SAMPLING ioctl call arguments 5-262
5-198 FCAN_SET_PRESCALER ioctl call arguments 5-263
5-199 FCAN_SET_RJW ioctl call arguments 5-264
5-200 FCAN_SET_PROP_SEG ioctl call arguments. 5-265
5-201 FCAN_SET_PHASE_SEG1, FCAN_SET_PHASE_SEG2 ioctl call arguments . . 5-266
5-202 FCAN_UNLOCK_ALL_MB ioctl call arguments. 5-267
5-203 FCAN_SET_MAXMB ioctl call arguments. 5-268
5-204 FCAN_READ_ERR_AND_STATUS ioctl call arguments. 5-269
5-205 FCAN_CLEAR_BOFF_INT ioctl call arguments 5-270
5-206 FCAN_CLEAR_ERR_INT ioctl call arguments 5-271
5-207 FCAN_CLEAR_WAKE_INT ioctl call arguments 5-272
5-208 FCAN_CLEAR_INT ioctl call arguments 5-273
5-209 FCAN_MBINT_ENABLE ioctl call arguments. 5-274
5-210 FCAN_MBINT_DISABLE ioctl call arguments 5-275
5-211 FCAN_READ_MBINT_FLAGS ioctl call arguments. 5-276
5-212 FCAN_CLEAR_MBINT_FLAGS ioctl call arguments. 5-277
5-213 FCAN_SET_RXGMASK ioctl call arguments 5-278
5-214 FCAN_SET_RXGMASK_RAW ioctl call arguments. 5-279
5-215 FCAN_SET_RX14MASK, FCAN_SET_RX15MASK ioctl call arguments 5-280
5-216 FCAN_SET_RX14MASK_RAW, FCAN_SET_RX15MASK_RAW ioctl call arguments5-281
5-217 FCAN_GET_RX_ERR_COUNT ioctl call arguments 5-282
5-218 FCAN_GET_TX_ERR_COUNT ioctl call arguments. 5-283
5-219 FCAN_GET_MB_MODULE ioctl call arguments 5-284
5-220 FCANMB_GET_ID ioctl call arguments. 5-285
5-221 FCANMB_GET_ID_RAW ioctl call arguments 5-286
5-222 FCANMB_GET_LEN ioctl call arguments 5-287
5-223 FCANMB_GET_DATAPTR ioctl call arguments. 5-288
5-224 FCANMB_GET_CODE ioctl call arguments 5-289
5-225 FCANMB_GET_TIMESTAMP ioctl call arguments 5-290
5-226 FCANMB_GET_TIMESTAMP8 ioctl call arguments 5-291
5-227 FCANMB_SET_ID ioctl call arguments 5-292
5-228 FCANMB_SET_ID_V ioctl call arguments. 5-294
5-229 FCANMB_SET_ID_RAW ioctl call arguments 5-295
5-230 FCANMB_SET_RTR ioctl call arguments 5-296
5-231 FCANMB_SET_LEN ioctl call arguments 5-297

5-232 FCANMB_SET_CODE ioctl call arguments 5-298
5-233 GPIO Module Base Address 5-307
5-234 GPIO Configuration Items for appconfig.h 5-308
5-235 GPIO Driver Arguments - ioctl 5-309
5-236 ioctl commands 5-309
5-237 GPIO_INIT ioctl call arguments 5-312
5-238 GPIO_INIT_ALL ioctl call arguments 5-313
5-239 GPIO_SETAS_GPIO ioctl call arguments 5-314
5-240 GPIO_SETAS_PERIPHERAL ioctl call arguments 5-315
5-241 GPIO_SETAS_INPUT ioctl call arguments 5-316
5-242 GPIO_SETAS_OUTPUT ioctl call arguments 5-317
5-243 GPIO_INT_DISABLE ioctl call arguments 5-318
5-244 GPIO_INT_ENABLE ioctl call arguments 5-319
5-245 GPIO_PULLUP_DISABLE ioctl call arguments 5-320
5-246 GPIO_PULLUP_ENABLE ioctl call arguments 5-321
5-247 GPIO_CLEAR_SW_INT_PENDING ioctl call arguments 5-322
5-248 GPIO_SW_INT_ASSERT ioctl call arguments 5-323
5-249 GPIO_INT_DETECTION_ACTIVE_HIGH ioctl call arguments 5-324
5-250 GPIO_INT_DETECTION_ACTIVE_LOW ioctl call arguments 5-325
5-251 GPIO_CLEAR_INT_PENDING ioctl call arguments 5-326
5-252 GPIO_SET_PIN ioctl call arguments 5-327
5-253 GPIO_CLEAR_PIN ioctl call arguments 5-328
5-254 GPIO_TOGGLE_PIN ioctl call arguments 5-329
5-255 GPIO_READ_DATA ioctl call arguments 5-330
5-256 GPIO_WRITE_DATA ioctl call arguments 5-331
5-257 GPIO_READ_INT_PENDING_REG ioctl call arguments 5-332
5-258 GPIO_GET_INT_PENDING_FLAG ioctl call arguments 5-333
5-259 GPIO_TEST_INT_PENDING ioctl call arguments 5-334
5-260 GPIO_SETAS_PUSHPULL ioctl call arguments 5-335
5-261 GPIO_SETAS_OPENDRAIN ioctl call arguments 5-336
5-262 GPIO_READ_RAW_DATA ioctl call arguments 5-337
5-263 GPIO_SET_HIGH_DRIVE_STRENGTH ioctl call arguments 5-338
5-264 GPIO_SET_LOW_DRIVE_STRENGTH ioctl call arguments 5-339
5-265 GPIO_SET_LOW_PASS_FILTER_ENABLE ioctl call arguments 5-340
5-266 GPIO_SET_LOW_PASS_FILTER_DISABLE ioctl call arguments 5-341
5-267 GPIO_SET_SLEW_RATE_FILTER_ENABLE ioctl call arguments 5-342
5-268 GPIO_SET_SLEW_RATE_FILTER_DISABLE ioctl call arguments 5-343
5-269 EVMs configuration 5-344
5-270 ADC Module Base Address 5-349

5-271 ADC Configuration Items for appconfig.h 5-350
5-272 ADC Driver Arguments - ioctl 5-352
5-273 ioctl commands 5-353
5-274 ADC_INIT ioctl call arguments 5-360
5-275 ADC_START ioctl call arguments 5-361
5-276 ADC_STOP ioctl call arguments 5-362
5-277 ADC_SYNC ioctl call arguments 5-363
5-278 ADC_SIMULT ioctl call arguments 5-364
5-279 ADC_SET_DIVISOR ioctl call arguments 5-365
5-280 ADC_SET_CHANNEL_CONFIG ioctl call arguments 5-366
5-281 ADC_SET_SCAN_MODE ioctl call arguments 5-368
5-282 ADC_SET_LIST_SAMPLEx ioctl call arguments 5-369
5-283 ADC_WRITE_SAMPLE_DISABLE ioctl call arguments 5-370
5-284 ADC_WRITE_ZERO_CROSS_CNTRL ioctl call arguments 5-371
5-285 ADC_ZERO_CROSS_CH0 ioctl call arguments 5-372
5-286 ADC_END_OF_SCAN_INT ioctl call arguments 5-373
5-287 ADC_ZERO_CROSS_INT ioctl call arguments 5-374
5-288 ADC_LOW_LIMIT_INT ioctl call arguments 5-375
5-289 ADC_LOW_LIMIT_INT ioctl call arguments 5-376
5-290 ADC_INT_ENABLE ioctl call arguments 5-377
5-291 ADC_INT_DISABLE ioctl call arguments 5-378
5-292 ADC_TEST_INT_ENABLED ioctl call arguments 5-379
5-293 ADC_READ_SAMPLE ioctl call arguments 5-380
5-294 ADC_READ_ALL_SAMPLES ioctl call arguments 5-381
5-295 ADC_READ_STATUS ioctl call arguments 5-382
5-296 ADC_READ_LIMIT_STATUS ioctl call arguments 5-384
5-297 ADC_READ_ZERO_CROSS_STATUS ioctl call arguments 5-385
5-298 ADC_GET_STATUS_CIP ioctl call arguments 5-386
5-299 ADC_GET_STATUS_EOSI ioctl call arguments 5-387
5-300 ADC_GET_STATUS_ZCI ioctl call arguments 5-388
5-301 ADC_GET_STATUS_LLMTI ioctl call arguments 5-389
5-302 ADC_GET_STATUS_HLMTI ioctl call arguments 5-390
5-303 ADC_GET_STATUS_RDY ioctl call arguments 5-391
5-304 ADC_GET_LIMIT_STATUS_LLS ioctl call arguments 5-392
5-305 ADC_GET_LIMIT_STATUS_HLS ioctl call arguments 5-393
5-306 ADC_GET_ZERO_CROSS_STATUS_ZCS ioctl call arguments 5-394
5-307 ADC_CLEAR_STATUS_EOSI ioctl call arguments 5-395
5-308 ADC_CLEAR_STATUS_LLMTI ioctl call arguments 5-396
5-309 ADC_CLEAR_STATUS_HLMTI ioctl call arguments 5-397

5-310 ADC_CLEAR_STATUS_ZCI ioctl call arguments 5-398
5-311 ADC_CLEAR_LIMIT_STATUS_LLS ioctl call arguments 5-399
5-312 ADC_CLEAR_LIMIT_STATUS_HLS ioctl call arguments. 5-400
5-313 ADC_CLEAR_LIMIT_STATUS_BITS ioctl call arguments 5-401
5-314 ADC_CLEAR_ZERO_CROSS_STATUS_ZCS ioctl call arguments. 5-402
5-315 ADC_WRITE_OFFSETx ioctl call arguments 5-403
5-316 ADC_WRITE_LOW_LIMITx ioctl call arguments. 5-404
5-317 ADC_WRITE_LOW_LIMITx ioctl call arguments. 5-405
5-318 ADC_READ_LOW_LIMIT ioctl call arguments 5-406
5-319 ADC_READ_HIGH_LIMIT ioctl call arguments 5-407
5-320 ADC_READ_OFFSET ioctl call arguments 5-408
5-321 ADC_POWER_DOWN ioctl call arguments. 5-409
5-322 ADC_POWER_UP ioctl call arguments 5-410
5-323 ADC_POWER_SAVE_MODE ioctl call arguments 5-411
5-324 ADC_SET_POWER_UP_DELAY ioctl call arguments 5-412
5-325 ADC_GET_POWER_STATUS ioctl call arguments. 5-413
5-326 ADC_READ_POWER_CONTROL_REG ioctl call arguments 5-414
5-327 ADC_CALIB_ENABLE ioctl call arguments 5-415
5-328 ADC_CALIB_DISABLE ioctl call arguments. 5-416
5-329 ADC_SET_CONVERTER0_CALIB_REF ioctl call arguments. 5-417
5-330 ADC_SET_CONVERTER1_CALIB_REF ioctl call arguments. 5-418
5-331 ADC_POWER_SAVE_MODE ioctl call arguments 5-419
5-332 ADC_POWER_SAVE_MODE ioctl call arguments 5-420
5-333 ADC_SET_VREFL_SOURCE ioctl call arguments 5-421
5-334 ADC_SET_VREFH_SOURCE ioctl call arguments 5-422
5-335 ADC_SET_VREFH0_SOURCE ioctl call arguments 5-423
5-336 ADC_SET_VREFL0_SOURCE ioctl call arguments 5-424
5-337 ADC_SET_VREFH1_SOURCE ioctl call arguments 5-425
5-338 ADC_SET_VREFL1_SOURCE ioctl call arguments 5-426
5-339 ADC_SET_CALIB_SOURCE ioctl call arguments. 5-427
5-340 ADC Module Base Address 5-433
5-341 ADC Configuration Items for appconfig.h. 5-434
5-342 ADC Driver Arguments - ioctl 5-435
5-343 ioctl commands. 5-435
5-344 ADC_INIT ioctl call arguments 5-438
5-345 ADC_SET_CONVERSION_MODE_A ioctl call arguments 5-439
5-346 ADC_SET_CONVERSION_MODE_B ioctl call arguments 5-440
5-347 ADC_SET_INPUT_CHANNEL_A ioctl call arguments. 5-441
5-348 ADC_SET_INPUT_CHANNEL_B ioctl call arguments. 5-442

5-349 ADC_TEST_CONVERSION_COMPLETE_A ioctl call arguments 5-443
5-350 ADC_TEST_CONVERSION_COMPLETE_B ioctl call arguments 5-444
5-351 ADC_TEST_CONVERSION_ACTIVE ioctl call arguments 5-445
5-352 ADC_SET_TRIGGER_MODE ioctl call arguments 5-446
5-353 ADC_SET_CONVERSION_CLOCK_OUT ioctl call arguments. 5-447
5-354 ADC_SET_VOLTAGE_REFERENCE ioctl call arguments. 5-448
5-355 ADC_READ_SAMPLE_A ioctl call arguments 5-449
5-356 ADC_READ_SAMPLE_B ioctl call arguments 5-450
5-357 ADC_SET_DIVIDER ioctl call arguments 5-451
5-358 ADC_SET_SAMPLE_TIME ioctl call arguments. 5-452
5-359 ADC_SET_RESOLUTION ioctl call arguments 5-453
5-360 ADC_SET_CLOCK_INPUT ioctl call arguments. 5-454
5-361 PGA Module Base Address 5-457
5-362 PGA Configuration Items for appconfig.h 5-458
5-363 PGA Driver Arguments - ioctl 5-458
5-364 ioctl commands. 5-459
5-365 PGA_INIT ioctl call arguments 5-461
5-366 PGA_SET_TRIGGER_MODE ioctl call arguments 5-462
5-367 PGA_SET_GAIN ioctl call arguments. 5-463
5-368 PGA_SET_GAIN_SH ioctl call arguments 5-464
5-369 PGA_SET_GAIN_DIFF ioctl call arguments 5-465
5-370 PGA_SET_GAIN_DIFF_2 ioctl call arguments 5-466
5-371 PGA_SET_POWER_MODE ioctl call arguments. 5-467
5-372 PGA_ENABLE_MODULE ioctl call arguments. 5-468
5-373 PGA_DISABLE_MODULE ioctl call arguments 5-469
5-374 PGA_SET_SH_BYPASS ioctl call arguments. 5-470
5-375 PGA_SET_CALIBRATION_MODE ioctl call arguments 5-471
5-376 PGA_SET_CHARGE_PUMP_DIV ioctl call arguments 5-472
5-377 PGA_SET_SW_TRIGGER ioctl call arguments 5-473
5-378 PGA_SET_DIVIDER ioctl call arguments 5-474
5-379 PGA_SET_CLK_GS ioctl call arguments 5-475
5-380 PGA_TEST_RUNNING ioctl call arguments 5-476
5-381 PGA_TEST_STARTUP_COMPLETE ioctl call arguments 5-477
5-382 PDB Module Base Address 5-479
5-383 PDB Configuration Items for appconfig.h 5-480
5-384 PDB Driver Arguments - ioctl 5-480
5-385 ioctl commands. 5-481
5-386 PDB_INIT ioctl call arguments 5-483
5-387 PDB_SET_PRESCALER ioctl call arguments 5-484

5-388 PDB_SET_TRIGGER_A_OUT ioctl call arguments. 5-485
5-389 PDB_SET_TRIGGER_B_OUT ioctl call arguments. 5-486
5-390 PDB_SET_CONTINUOUS_MODE ioctl call arguments 5-487
5-391 PDB_SET_SW_TRIGGER ioctl call arguments 5-488
5-392 PDB_SET_INPUT_TRIGGER ioctl call arguments 5-489
5-393 PDB_SET_TRIGGER_A_ENABLE ioctl call arguments. 5-490
5-394 PDB_SET_TRIGGER_A_DISABLE ioctl call arguments 5-491
5-395 PDB_SET_TRIGGER_B_ENABLE ioctl call arguments 5-492
5-396 PDB_SET_TRIGGER_B_DISABLE ioctl call arguments 5-493
5-397 PDB_WRITE_DELAYA ioctl call arguments. 5-494
5-398 PDB_WRITE_DELAYA ioctl call arguments. 5-495
5-399 PDB_WRITE_MOD ioctl call arguments 5-496
5-400 PDB_READ_COUNTER_REG ioctl call arguments 5-497
5-401 Quadrature Decoder Module Base Address 5-499
5-402 decoder_sState Data Structure Members 5-501
5-403 decoder_sEncScale Data Structure Members. 5-501
5-404 decoder_sEncSignals Data Structure Members 5-501
5-405 Quadrature Decoder Configuration Items for appconfig.h. 5-502
5-406 Quadrature Decoder Driver Arguments - ioctl 5-503
5-407 ioctl commands. 5-503
5-408 DEC_INIT ioctl call arguments 5-507
5-409 DEC_INT_ENABLE ioctl call arguments 5-508
5-410 DEC_INT_DISABLE ioctl call arguments 5-509
5-411 DEC_INT_REQUEST_CLEAR ioctl call arguments 5-510
5-412 DEC_CLEAR_HOME_INT_REQUEST ioctl call arguments 5-511
5-413 DEC_CLEAR_INDEX_PULSE_INT_REQUEST ioctl call arguments 5-512
5-414 DEC_CLEAR_WATCHDOG_INT_REQUEST ioctl call arguments. 5-513
5-415 DEC_HOME_TRIGGERED_INIT ioctl call arguments 5-514
5-416 DEC_HOME_EDGE ioctl call arguments 5-515
5-417 DEC_SOFTWARE_TRIGGERED_INIT ioctl call arguments 5-516
5-418 DEC_DIRECTION_COUNTING_ENABLE ioctl call arguments 5-517
5-419 DEC_SINGLE_PHASE_COUNT ioctl call arguments 5-518
5-420 DEC_INDEX_TRIGGERED_INIT ioctl call arguments. 5-519
5-421 DEC_INDEX_EDGE ioctl call arguments. 5-520
5-422 DEC_WATCHDOG ioctl call arguments. 5-521
5-423 DEC_SWITCH_MATRIX ioctl call arguments. 5-522
5-424 Switch Matrix Mode 5-522
5-425 DEC_WRITE_FILTER ioctl call arguments 5-523
5-426 DEC_WRITE_WATCHDOG_TIMEOUT ioctl call arguments 5-524

5-427 DEC_READ_POSITION_DIFFERENCE ioctl call arguments. 5-525
5-428 DEC_READ_REVOLUTION ioctl call arguments 5-526
5-429 DEC_WRITE_REVOLUTION ioctl call arguments 5-527
5-430 DEC_READ_POSITION ioctl call arguments. 5-528
5-431 DEC_WRITE_POSITION ioctl call arguments 5-529
5-432 DEC_WRITE_INIT_STATE ioctl call arguments. 5-530
5-433 DEC_READ_MONITOR_REG ioctl call arguments 5-531
5-434 DEC_GET_RAW_ENCSIGNALS ioctl call arguments 5-532
5-435 DEC_GET_FILTERED_ENCSIGNALS ioctl call arguments 5-533
5-436 DEC_READ_HOLD_DATA_REGS ioctl call arguments. 5-534
5-437 DEC_READ_CONTROL_REG ioctl call arguments 5-535
5-438 DEC_CALCULATE_SCALE_COEF ioctl call arguments 5-536
5-439 DEC_GET_SCALED_POSITION ioctl call arguments 5-537
5-440 DEC_GET_SCALED_POSITION_DIFFERENCE ioctl call arguments . . . 5-538
5-441 DEC_HOME_INT ioctl call arguments 5-539
5-442 DEC_INDEX_PULSE_INT ioctl call arguments. 5-540
5-443 DEC_WATCHDOG_INT ioctl call arguments 5-541
5-444 EVMs configuration 5-542
5-445 PWM Module Base Address. 5-547
5-446 pwm_sComplementaryValues Data Structure Members 5-549
5-447 pwm_sIndependentValues Data Structure Members 5-549
5-448 pwm_sOutputControl Data Structure Members 5-550
5-449 pwm_sUpdateValueSetVlmode Data Structure Members 5-550
5-450 pwm_sChannelControl Data Structure Members 5-550
5-451 PWM Configuration Items for appconfig.h 5-550
5-452 PWM Driver Arguments - ioctl 5-552
5-453 ioctl commands 5-552
5-454 PWM_INIT ioctl call arguments 5-561
5-455 PWM_SET_RELOAD_FREQUENCY ioctl call arguments 5-562
5-456 PWM_HALF_CYCLE_RELOAD ioctl call arguments. 5-563
5-457 PWM_SET_CURRENT_POLARITY ioctl call arguments. 5-564
5-458 PWM_SET_PRESCALER ioctl call arguments. 5-566
5-459 PWM Prescaler 5-566
5-460 PWM_RELOAD_INT ioctl call arguments 5-568
5-461 PWM_SET_CURRENT_SENSING ioctl call arguments 5-569
5-462 Correction Method Selection 5-569
5-463 PWM_DEVICE ioctl call arguments 5-570
5-464 PWM_CLEAR_RELOAD_FLAG ioctl call arguments. 5-571
5-465 PWM_LOAD_OK ioctl call arguments 5-572

5-466 PWM_FAULT_INT_ENABLE ioctl call arguments 5-573
5-467 PWM_FAULT_INT_DISABLE ioctl call arguments 5-574
5-468 PWM_SET_AUTOMATIC_FAULT_CLEAR ioctl call arguments 5-575
5-469 PWM_SET_MANUAL_FAULT_CLEAR ioctl call arguments 5-576
5-470 PWM_CLEAR_FAULT_FLAG ioctl call arguments 5-577
5-471 PWM_OUTPUT_PAD ioctl call arguments. 5-578
5-472 PWM_OUTPUT_SOFTWARE_CONTROL ioctl call arguments 5-579
5-473 PWM_OUTPUT_CONTROL ioctl call arguments 5-580
5-474 PWM_SET_MODULO ioctl call arguments 5-581
5-475 PWM_GET_MODULO ioctl call arguments. 5-582
5-476 PWM_SET_DEADTIME ioctl call arguments 5-583
5-477 PWM_SET_DEADTIME_0 ioctl call arguments 5-584
5-478 PWM_SET_DEADTIME_1 ioctl call arguments 5-585
5-479 PWM_WRITE_DISABLE_MAPPING_REG1 ioctl call arguments. 5-586
5-480 PWM Pin - Fault Mapping Matrix 1. 5-586
5-481 PWM_WRITE_DISABLE_MAPPING_REG2 ioctl call arguments. 5-588
5-482 PWM Pin - Fault Mapping Matrix 2. 5-588
5-483 PWM_SET_ALIGNMENT ioctl call arguments 5-589
5-484 PWM_SET_NEG_TOP_SIDE_POLARITY ioctl call arguments. 5-590
5-485 PWM_SET_NEG_BOTTOM_SIDE_POLARITY ioctl call arguments 5-591
5-486 PWM_SET_INDEPENDENT_OPERATION ioctl call arguments. 5-592
5-487 PWM_SET_INDEPENDENT_MODE ioctl call arguments 5-593
5-488 PWM_SET_COMPLEMENTARY_MODE ioctl call arguments 5-594
5-489 PWM_SET_WRITE_PROTECT ioctl call arguments. 5-595
5-490 PWM_HARDWARE_ACCELERATION ioctl call arguments. 5-596
5-491 PWM_SET_CHANNEL_MASK ioctl call arguments. 5-597
5-492 PWM_SET_LOAD_MODE ioctl call arguments 5-598
5-493 PWM_SET_SWAP ioctl call arguments 5-599
5-494 PWM_WRITE_VALUE_REG_x ioctl call arguments 5-600
5-495 PWM_WRITE_VALUE_REGS_COMPL ioctl call arguments 5-601
5-496 PWM_WRITE_VALUE_REGS_INDEP ioctl call arguments 5-602
5-497 PWM_READ_FAULT_STATUS_REG ioctl call arguments 5-603
5-498 PWM_READ_COUNTER_REG ioctl call arguments. 5-604
5-499 PWM_READ_CONTROL_REG ioctl call arguments. 5-605
5-500 PWM_READ_PORT_REG ioctl call arguments 5-606
5-501 PWM_GET_CURRENT_STATUS_INPUTS ioctl call arguments. 5-607
5-502 PWM_GET_FAULT_INPUTS ioctl call arguments 5-608
5-503 PWM_GET_FAULT_INPUT_x ioctl call arguments 5-609
5-504 PWM_SOFTWARE_OUTPUTS_CONTROL ioctl call arguments 5-610

5-505 PWM_UPDATE_VALUE_REG_x ioctl call arguments 5-611
5-506 PWM_UPDATE_VALUE_REGS_COMPL ioctl call arguments 5-613
5-507 PWM_UPDATE_VALUE_REGS_INDEP ioctl call arguments 5-615
5-508 PWM_UPDATE_VALUE_SET_VLMODE ioctl call arguments 5-616
5-509 PWM_SET_MASK_SWAP ioctl call arguments. 5-618
5-510 PWM_DEBUG_OPERATION ioctl call arguments 5-619
5-511 PWM_WAIT_OPERATION ioctl call arguments 5-620
5-512 PWM_MASK_SWAP_OPERATION ioctl call arguments 5-621
5-513 PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION ioctl call arguments . 5-622
5-514 PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION ioctl call arguments. . 5-623
5-515 PWM_READ_INTERNAL_CORRECTION_CONTROL_REG ioctl call arguments5-624
5-516 PWM_SET_SOURCE_0 ioctl call arguments 5-625
5-517 PWM_SET_SOURCE_1 ioctl call arguments 5-626
5-518 PWM_SET_SOURCE_2 ioctl call arguments 5-627
5-519 PWM_SET_COMPARE_INVERT_x ioctl call arguments 5-628
5-520 PWM_SET_ACTIVE_HIGH_FAULTS ioctl call arguments 5-629
5-521 PWM_SET_ACTIVE_LOW_FAULTS ioctl call arguments 5-630
5-522 PWM_DISABLE_SYNC ioctl call arguments. 5-631
5-523 PWM_ENABLE_SYNC_OUT ioctl call arguments 5-632
5-524 PWM_ENABLE_SYNC_IN ioctl call arguments 5-633
5-525 PWM_WRITE_FILTx_REG ioctl call arguments 5-634
5-526 PWM_WRITE_FILTx_REG ioctl call arguments 5-635
5-527 PWM_SET_PULSE_EDGE_CONTROL_x ioctl call arguments 5-636
5-528 SCI Module Base Address 5-645
5-529 SCI Configuration Items for appconfig.h 5-646
5-530 SCI Operation Items for appconfig.h 5-646
5-531 Built-in Software Layer Configuration Items for appconfig.h 5-647
5-532 Buffer Space Monitoring Configuration Items. 5-648
5-533 SCI Driver Arguments - ioctl 5-649
5-534 SCI Driver Arguments - read/write 5-649
5-535 ioctl commands. 5-650
5-536 read/write modes. 5-653
5-537 SCI_INIT ioctl call arguments 5-654
5-538 SCI_SET_BAUDRATE ioctl call arguments. 5-655
5-539 SCI_OPERATING_MODE ioctl call arguments 5-656
5-540 SCI_TRANSMITTER ioctl call arguments 5-657
5-541 SCI_RECEIVER ioctl call arguments 5-658

5-542 SCI_WAKEUP_CONDITION ioctl call arguments 5-659
5-543 SCI_DATA_FORMAT ioctl call arguments 5-660
5-544 SCI_PARITY ioctl call arguments 5-661
5-545 SCI_DATA_POLARITY ioctl call arguments. 5-662
5-546 SCI_STOP_IN_WAIT ioctl call arguments 5-663
5-547 SCI_SEND_BREAK ioctl call arguments 5-664
5-548 SCI_WAIT ioctl call arguments 5-665
5-549 SCI_WAKEUP ioctl call arguments. 5-666
5-550 SCI_INT_ENABLE ioctl call arguments 5-667
5-551 SCI_INT_DISABLE ioctl call arguments 5-668
5-552 SCI_GET_STATUS_REG ioctl call arguments. 5-669
5-553 SCI_CLEAR_STATUS_REG ioctl call arguments 5-670
5-554 SCI_TEST_STATUS_REG ioctl call arguments 5-671
5-555 SCI_READ_DATA ioctl call arguments 5-672
5-556 SCI_WRITE_DATA ioctl call arguments 5-673
5-557 SCI_READ_CONTROL_REG ioctl call arguments 5-674
5-558 SCI_WRITE_CONTROL_REG ioctl call arguments 5-675
5-559 SCI_GET_TX_EMPTY ioctl call arguments 5-676
5-560 SCI_GET_TX_IDLE ioctl call arguments 5-677
5-561 SCI_GET_RX_FULL ioctl call arguments 5-678
5-562 SCI_GET_RX_IDLE ioctl call arguments 5-679
5-563 SCI_GET_RX_ERROR ioctl call arguments 5-680
5-564 SCI_GET_RX_OVERRUN ioctl call arguments 5-681
5-565 SCI_GET_RX_NOISE_ERROR ioctl call arguments 5-682
5-566 SCI_GET_RX_FRAMING_ERROR ioctl call arguments. 5-683
5-567 SCI_GET_RX_PARITY_ERROR ioctl call arguments. 5-684
5-568 SCI_GET_RX_ACTIVE ioctl call arguments 5-685
5-569 SCI_LIN_MODE ioctl call arguments 5-686
5-570 SCI_QUEUED_MODE ioctl call arguments 5-687
5-571 SCI_SET_TXEMPTY_CONDITION ioctl call arguments 5-688
5-572 SCI_SET_RXFULL_CONDITION ioctl call arguments. 5-689
5-573 SCI_CAN_READ_DATA ioctl call arguments 5-690
5-574 SCI_CAN_WRITE_DATA ioctl call arguments 5-691
5-575 read function call arguments. 5-692
5-576 interrupt routines of the SCI driver for the read function in non-blocking mode. . . 5-693
5-577 write function call arguments 5-694
5-578 interrupt routines of the SCI driver for the write function in non-blocking mode . . 5-694

5-579 SCI_CLEAR_EXCEPTION ioctl call arguments 5-695
5-580 SCI0_GET_STATUS ioctl call arguments. 5-696
5-581 SCI_WRITE_CANCEL ioctl call arguments. 5-697
5-582 SCI_READ_CANCEL ioctl call arguments. 5-698
5-583 SCI_BUFFERED_RX ioctl call arguments 5-699
5-584 SCI_BUFFERED_TX ioctl call arguments 5-700
5-585 SCI_GET_RX_CHARS_READY ioctl call arguments 5-701
5-586 SCI_GET_RX_BUFFER_FREESPACE ioctl call arguments. 5-702
5-587 SCI_GET_TX_BUFFER_FREESPACE ioctl call arguments 5-703
5-588 SCI_SEND_XON ioctl call arguments. 5-704
5-589 SCI_SEND_XOFF ioctl call arguments. 5-705
5-590 SPI Module Base Address 5-713
5-591 SPI Configuration Items for appconfig.h 5-714
5-592 SPI Operation Items for appconfig.h 5-714
5-593 Built-in Software Layer Configuration Items. 5-715
5-594 SPI Driver Arguments - ioctl 5-716
5-595 SPI Driver Arguments - read/write. 5-716
5-596 ioctl commands. 5-717
5-597 read/write modes. 5-720
5-598 SPI_INIT ioctl call arguments 5-721
5-599 SPI_DEVICE ioctl call arguments 5-722
5-600 SPI_SET_MODE ioctl call arguments. 5-723
5-601 SPI_SET_ORDER ioctl call arguments 5-724
5-602 SPI_SET_CLOCK_POLARITY ioctl call arguments 5-725
5-603 SPI_SET_CLOCK_PHASE ioctl call arguments. 5-726
5-604 SPI_SET_MODE_FAULT ioctl call arguments 5-727
5-605 SPI_SET_TX_DATA_SIZE ioctl call arguments 5-728
5-606 SPI_SET_WIRED_OR_MODE ioctl call arguments. 5-729
5-607 SPI_SET_BAUD_DIV ioctl call arguments. 5-730
5-608 SPI_RX_FULL_INT ioctl call arguments 5-731
5-609 SPI_TX_EMPTY_INT ioctl call arguments. 5-732
5-610 SPI_ERROR_INT ioctl call arguments 5-733
5-611 SPI_INT_ENABLE ioctl call arguments 5-734
5-612 SPI_INT_DISABLE ioctl call arguments. 5-735
5-613 SPI_INT_SELECT ioctl call arguments. 5-736
5-614 SPI_WRITE_DATA ioctl call arguments 5-737
5-615 SPI_READ_DATA ioctl call arguments 5-738
5-616 SPI_GET_TX_EMPTY ioctl call arguments 5-739
5-617 SPI_GET_RX_FULL ioctl call arguments. 5-740

5-618 SPI_GET_RX_OVERFLOW ioctl call arguments. 5-741
5-619 SPI_READ_CONTROL_REG ioctl call arguments 5-742
5-620 SPI_WRITE_CONTROL_REG ioctl call arguments. 5-743
5-621 SPI_GET_MODE_FAULT ioctl call arguments 5-744
5-622 SPI_GET_ERROR ioctl call arguments. 5-745
5-623 SPI_CLEAR_MODE_FAULT ioctl call arguments 5-746
5-624 SPI_MULT_BAUD_DIV ioctl call arguments 5-747
5-625 SPI_TEST_SS_INPUT ioctl call arguments 5-748
5-626 SPI_SET_SS_MODE ioctl call arguments. 5-749
5-627 SPI_SET_SS_OUTPUT ioctl call arguments. 5-751
5-628 SPI_SET_SS_WIRED_OR_MODE ioctl call arguments 5-752
5-629 SPI_OVERRIDE_SS_INPUT ioctl call arguments 5-753
5-630 SPI_QUEUED_MODE ioctl call arguments 5-754
5-631 SPI_SET_TXEMPTY_CONDITION ioctl call arguments 5-755
5-632 SPI_SET_RXFULL_CONDITION ioctl call arguments 5-756
5-633 SPI_CAN_READ_DATA ioctl call arguments 5-757
5-634 SPI_CAN_WRITE_DATA ioctl call arguments 5-758
5-635 read function call arguments. 5-759
5-636 Interrupt routines of the SPI driver for the read function in non-blocking mode . . . 5-759
5-637 write function call arguments 5-760
5-638 Interrupt routines of the SPI driver for the write function in non-blocking mode . . 5-760
5-639 SPI_CLEAR_EXCEPTION ioctl call arguments. 5-761
5-640 SPI_GET_STATUS ioctl call arguments 5-762
5-641 SPI_WRITE_CANCEL ioctl call arguments 5-763
5-642 SPI_READ_CANCEL ioctl call arguments 5-764
5-643 IIC Module Base Address. 5-774
5-644 IIC Configuration Items for appconfig.h MC56F801x. 5-775
5-645 IIC Configuration Items for appconfig.h MC56F800x. 5-776
5-646 IIC Driver Arguments - ioctl 5-777
5-647 IIC Module ioctl Commands 5-777
5-648 IIC_INIT ioctl call arguments. 5-781
5-649 IIC_SET_ADDRESS ioctl call arguments 5-782
5-650 IIC_GET_ADDRESS ioctl call arguments. 5-783
5-651 IIC_SET_PRESCALER ioctl call arguments. 5-784
5-652 IIC_I_BUS ioctl call arguments 5-785
5-653 IIC_I_BUS_INT ioctl call arguments. 5-786
5-654 IIC_MASTER_SLAVE_MODE ioctl call arguments 5-787

5-655 IIC_GET_MASTER_MODE ioctl call arguments 5-788
5-656 IIC_TX_RX_MODE ioctl call arguments 5-789
5-657 IIC_GET_TX_MODE ioctl call arguments 5-790
5-658 IIC_TX_ACK ioctl call arguments 5-791
5-659 IIC_REPEAT_START ioctl call arguments 5-792
5-660 IIC_WRITE_CONTROL_REG ioctl call arguments 5-793
5-661 IIC_READ_CONTROL_REG ioctl call arguments 5-794
5-662 IIC_READ_DATA ioctl call arguments 5-795
5-663 IIC_WRITE_DATA ioctl call arguments 5-796
5-664 IIC_SET_NOISE_FILTER ioctl call arguments 5-797
5-665 IIC_READ_STATUS_REG ioctl call arguments 5-798
5-666 IIC_TEST_STATUS_REG ioctl call arguments 5-799
5-667 IIC_CLEAR_ARBITRATION_LOST ioctl call arguments 5-800
5-668 IIC_CLEAR_I_BUS_INT ioctl call arguments 5-801
5-669 IIC_SET_GENERAL_CALL_ADDRESS ioctl call arguments 5-802
5-670 IIC_SET_ADDRESS_EXTENSION_MODE ioctl call arguments 5-803
5-671 IIC_SET_ADDRESS_EXTENSION ioctl call arguments 5-804
5-672 IIC_SET_10BIT_ADDRESS ioctl call arguments 5-805
5-673 IIC_GET_10BIT_ADDRESS ioctl call arguments 5-806
5-674 IIC_SET_FAST_ACK_NACK ioctl call arguments 5-807
5-675 IIC_SET_SMBUS_RESPONSE_ADDRESS ioctl call arguments 5-808
5-676 IIC_SET_SECOND_IIC_ADDRESS ioctl call arguments 5-809
5-677 IIC_SET_TIME_OUT_CLOCK ioctl call arguments 5-810
5-678 IIC_TEST_TIMEOUT_FLAGS ioctl call arguments 5-811
5-679 IIC_CLEAR_LOW_TIMEOUT_FLAG ioctl call arguments 5-812
5-680 IIC_SET_SMBUS_ADDRESS ioctl call arguments 5-813
5-681 IIC_GET_SMBUS_ADDRESS ioctl call arguments 5-814
5-682 IIC_WRITE_SCL_LOW_TIMEOUT ioctl call arguments 5-815
5-683 IIC_READ_SCL_LOW_TIMEOUT ioctl call arguments 5-816
5-684 IIC Module Base Address 5-819
5-685 IIC Configuration Items for appconfig.h 5-820
5-686 IIC Driver Arguments - ioctl 5-821
5-687 ioctl commands 5-821
5-688 IIC_INIT ioctl call arguments 5-826
5-689 IIC_MODULE ioctl call arguments 5-827
5-690 IIC_TRY_GRACEFUL_SHUTDOWN ioctl call arguments 5-828
5-691 IIC_SLAVE_OPERATION ioctl call arguments 5-829
5-692 IIC_MASTER_OPERATION ioctl call arguments 5-830
5-693 IIC_USE_REPEATED_START ioctl call arguments 5-831

5-694 IIC_SET_ADDRESS ioctl call arguments 5-832
5-695 IIC_SET_SPEED_MODE ioctl call arguments 5-833
5-696 IIC_SET_xxxCNT ioctl call arguments 5-834
5-697 IIC_CAN_INITIATE_TRANSACTION ioctl call arguments. 5-835
5-698 IIC_INITIATE_TRANSACTION ioctl call arguments 5-836
5-699 IIC_INITIATE_SB_TRANSACTION ioctl call arguments 5-837
5-700 IIC_INITIATE_GC_TRANSACTION ioctl call arguments 5-838
5-701 IIC_MASTER_TRANSACTION_ACTIVE ioctl call arguments 5-839
5-702 IIC_CAN_READ_DATA ioctl call arguments 5-840
5-703 IIC_READ_DATA ioctl call arguments. 5-841
5-704 IIC_CAN_WRITE_DATA ioctl call arguments 5-842
5-705 IIC_WRITE_DATA ioctl call arguments. 5-843
5-706 IIC_CAN_REQUEST_DATA ioctl call arguments. 5-844
5-707 IIC_READ_REQUEST ioctl call arguments 5-845
5-708 IIC_INT_ENABLE ioctl call arguments 5-846
5-709 IIC_INT_DISABLE ioctl call arguments. 5-847
5-710 IIC_READ_INT_FLAGS ioctl call arguments 5-848
5-711 IIC_TEST_INT_FLAGS ioctl call arguments 5-849
5-712 IIC_CLEAR_EINT_FLAGS ioctl call arguments 5-850
5-713 IIC_CLEAR_GINT_FLAGS ioctl call arguments 5-851
5-714 IIC_CLEAR_TINT_FLAGS ioctl call arguments 5-852
5-715 IIC_CLEAR_SINT_FLAGS ioctl call arguments 5-853
5-716 IIC_CLEAR_ALL_INTS ioctl call arguments. 5-854
5-717 IIC_READ_TXABORT_SOURCE ioctl call arguments. 5-855
5-718 IIC_CLEAR_TXABORT_SOURCE ioctl call arguments. 5-856
5-719 IIC_READ_STATUS_FLAGS ioctl call arguments 5-857
5-720 IIC_TEST_STATUS_FLAGS ioctl call arguments 5-858
5-721 Temperature Sensor System Module Base Address. 5-861
5-722 Temperature Sensor System Configuration Items for appconfig.h 5-862
5-723 Temperature Sensor Driver Arguments - ioctl 5-862
5-724 ioctl commands. 5-863
5-725 TSENSOR_INIT ioctl call arguments 5-864
5-726 TSENSOR_DEVICE ioctl call arguments 5-865
5-727 TSENSOR_IS_POWERED_ON ioctl call arguments 5-866
5-728 Quad Timer Module Base Address. 5-869
5-729 Quad Timer Configuration Items for appconfig.h 5-870
5-730 Quad Timer Driver Arguments - ioctl 5-871
5-731 ioctl commands. 5-871
5-732 QT_INIT ioctl call arguments. 5-876

5-733 QT_SET_COUNT_MODE ioctl call arguments 5-877
5-734 QT_SET_PRIMARY_SOURCE ioctl call arguments 5-879
5-735 QT_SET_SECONDARY_SOURCE ioctl call arguments 5-880
5-736 QT_SET_COUNT_ONCE ioctl call arguments 5-881
5-737 QT_SET_COUNT_LENGTH ioctl call arguments 5-882
5-738 QT_SET_COUNT_DIRECTION ioctl call arguments 5-883
5-739 QT_CO_CHANNEL_INIT ioctl call arguments 5-884
5-740 QT_SET_OUTPUT_MODE ioctl call arguments 5-885
5-741 QT_CLEAR_FLAG ioctl call arguments 5-886
5-742 QT_CLEAR_COMPARE_FLAG ioctl call arguments 5-887
5-743 QT_READ_FLAG ioctl call arguments 5-888
5-744 QT_READ_COMPARE_FLAG ioctl call arguments 5-889
5-745 QT_INT_ENABLE ioctl call arguments 5-890
5-746 QT_INT_DISABLE ioctl call arguments 5-891
5-747 QT_SET_INPUT_POLARITY ioctl call arguments 5-892
5-748 QT_READ_EXT_INPUT_PIN ioctl call arguments 5-893
5-749 QT_SET_CAPTURE_MODE ioctl call arguments 5-894
5-750 QT_MASTER_MODE ioctl call arguments. 5-895
5-751 QT_EXT_OFLAG_FORCE ioctl call arguments. 5-896
5-752 QT_FORCE_OFLAG ioctl call arguments 5-897
5-753 QT_SET_OUTPUT_POLARITY ioctl call arguments 5-898
5-754 QT_OUTPUT_ON_EXT_PIN ioctl call arguments 5-899
5-755 QT_SET_LOAD_CONTROL1 ioctl call arguments 5-900
5-756 QT_SET_LOAD_CONTROL2 ioctl call arguments 5-901
5-757 QT_WRITE_CONTROL_REG ioctl call arguments 5-902
5-758 QT_WRITE_STATUS_CONTROL_REG ioctl call arguments 5-903
5-759 QT_WRITE_CMP_STATUS_CONTROL_REG ioctl call arguments 5-904
5-760 QT_WRITE_COMPARE_REG1 ioctl call arguments. 5-905
5-761 QT_WRITE_COMPARE_REG2 ioctl call arguments. 5-906
5-762 QT_WRITE_PRELOAD_COMPARE_REG1 ioctl call arguments 5-907
5-763 QT_WRITE_PRELOAD_COMPARE_REG2 ioctl call arguments 5-908
5-764 QT_WRITE_LOAD_REG ioctl call arguments. 5-909
5-765 QT_WRITE_COUNTER_REG ioctl call arguments 5-910
5-766 QT_READ_CONTROL_REG ioctl call arguments 5-911
5-767 QT_READ_STATUS_CONTROL_REG ioctl call arguments 5-912
5-768 QT_READ_CMP_STATUS_CONTROL_REG ioctl call arguments 5-913
5-769 QT_READ_COMPARE_REG1 ioctl call arguments 5-914
5-770 QT_READ_COMPARE_REG2 ioctl call arguments 5-915
5-771 QT_READ_LOAD_REG ioctl call arguments. 5-916

5-772 QT_READ_COUNTER_REG ioctl call arguments 5-917
5-773 QT_READ_COUNTER_REG ioctl call arguments 5-918
5-774 QT_READ_HOLD_REG ioctl call arguments. 5-919
5-775 QT0_MASS_ENABLE ioctl call arguments 5-920
5-776 QT0_MASS_DISABLE ioctl call arguments. 5-921
5-777 QT_WRITE_FILT_REG ioctl call arguments 5-922
5-778 QT_READ_FILT_REG ioctl call arguments 5-923
5-779 PIT Module Base Address 5-931
5-780 PIT Configuration Items for appconfig.h 5-932
5-781 PIT Driver Arguments - ioctl 5-933
5-782 ioctl commands. 5-933
5-783 PIT_INIT ioctl call arguments 5-935
5-784 PIT_COUNTER ioctl call arguments. 5-936
5-785 PIT_SLAVE_MODE ioctl call arguments 5-937
5-786 PIT_ROLLOVER_INT ioctl call arguments 5-938
5-787 PIT_CLEAR_ROLLOVER_INT ioctl call arguments. 5-939
5-788 PIT_SET_PRESCALER ioctl call arguments 5-940
5-789 PIT_WRITE_MODULO_REG ioctl call arguments 5-941
5-790 PIT_READ_MODULO_REG ioctl call arguments 5-942
5-791 PIT_READ_COUNTER_REG ioctl call arguments 5-943
5-792 CMP Module Base Address 5-945
5-793 CMP Configuration Items for appconfig.h. 5-946
5-794 CMP Driver Arguments - ioctl 5-946
5-795 ioctl commands. 5-947
5-796 CMP_INIT ioctl call arguments 5-949
5-797 CMP_MODULE ioctl call arguments 5-950
5-798 CMP_INVERT ioctl call arguments. 5-951
5-799 CMP_SELECT_POS_INPUT ioctl call arguments 5-952
5-800 CMP_SELECT_NEG_INPUT ioctl call arguments. 5-953
5-801 CMP_SELECT_EXPORT_OUTPUT ioctl call arguments 5-954
5-802 CMP_INT_ENABLE ioctl call arguments. 5-955
5-803 CMP_INT_DISABLE ioctl call arguments 5-956
5-804 CMP_READ_INT_FLAGS ioctl call arguments 5-957
5-805 CMP_CLEAR_INT_FLAGS ioctl call arguments. 5-958
5-806 CMP_READ_OUTPUT ioctl call arguments. 5-959
5-807 CMP_WRITE_FILT_REG ioctl call arguments 5-960
5-808 CMP_READ_FILT_REG ioctl call arguments 5-961
5-809 HSCMP Module Base Address. 5-963
5-810 CMP Configuration Items for appconfig.h. 5-964

5-811 CMP Driver Arguments - ioctl 5-965
5-812 ioctl commands 5-965
5-813 HSCMP_INIT ioctl call arguments 5-967
5-814 HSCMP_MODULE ioctl call arguments 5-968
5-815 HSCMP_SELECT_POS_INPUT ioctl call arguments. 5-969
5-816 HSCMP_SELECT_NEG_INPUT ioctl call arguments 5-970
5-817 HSCMP_SET_INVERT ioctl call arguments. 5-971
5-818 HSCMP_SET_SAMPLE ioctl call arguments 5-972
5-819 HSCMP_SET_WINDOWING ioctl call arguments. 5-973
5-820 HSCMP_SET_HIGH_SPEED ioctl call arguments 5-974
5-821 HSCMP_SET_OUTPUT_PIN ioctl call arguments 5-975
5-822 HSCMP_SET_OUTPUT_ACTIVE ioctl call arguments. 5-976
5-823 HSCMP_INT_RISING_EDGE ioctl call arguments 5-977
5-824 HSCMP_INT_FALLING_EDGE ioctl call arguments 5-978
5-825 HSCMP_TEST_INT_FLAGS ioctl call arguments 5-979
5-826 HSCMP_CLEAR_INT_FLAGS ioctl call arguments 5-980
5-827 HSCMP_READ_OUTPUT ioctl call arguments 5-981
5-828 HSCMP_WRITE_FILT_COUNTER ioctl call arguments 5-982
5-829 HSCMP_READ_FILT_COUNTER ioctl call arguments 5-983
5-830 HSCMP_WRITE_FILT_REG ioctl call arguments 5-984
5-831 HSCMP_READ_FILT_REG ioctl call arguments 5-985
5-832 DAC Module Base Address 5-987
5-833 DAC Configuration Items for appconfig.h 5-988
5-834 DAC Driver Arguments - ioctl 5-989
5-835 ioctl commands 5-989
5-836 DAC_INIT ioctl call arguments 5-991
5-837 DAC_MODULE ioctl call arguments 5-992
5-838 DAC_SET_DATA_FORMAT ioctl call arguments. 5-993
5-839 DAC_SET_SYNC_SOURCE ioctl call arguments 5-994
5-840 DAC_SET_AUTO_MODE ioctl call arguments 5-995
5-841 DAC_ENABLE_FILTER ioctl call arguments 5-996
5-842 DAC_DISABLE_FILTER ioctl call arguments 5-997
5-843 DAC_WRITE_CONTROL_REG ioctl call arguments 5-998
5-844 DAC_READ_CONTROL_REG ioctl call arguments 5-999
5-845 DAC_WRITE_DATA ioctl call arguments 5-1000
5-846 DAC_READ_DATA ioctl call arguments 5-1001
5-847 DAC_WRITE_STEP ioctl call arguments 5-1002
5-848 DAC_READ_STEP ioctl call arguments 5-1003
5-849 DAC_WRITE_MINVAL ioctl call arguments. 5-1004

5-850 DAC_READ_MINVAL ioctl call arguments. 5-1005
5-851 DAC_WRITE_MAXVAL ioctl call arguments 5-1006
5-852 DAC_READ_MAXVAL ioctl call arguments 5-1007
5-853 MSCAN Module Base Address 5-1009
5-854 ID-raw Registers with Standard CAN ID 5-1012
5-855 ID-raw Registers with Extended CAN ID 5-1012
5-856 MSCAN Configuration Items for appconfig.h 5-1013
5-857 MSCAN Driver Arguments - ioctl 5-1014
5-858 ioctl commands 5-1015
5-859 Message Buffer ioctl commands. 5-1019
5-860 MSCAN_INIT ioctl call arguments 5-1021
5-861 MSCAN_SOFT_RESET ioctl call arguments 5-1022
5-862 MSCAN_DEVICE ioctl call arguments 5-1023
5-863 MSCAN_WAKEUP_FILTER ioctl call arguments 5-1024
5-864 MSCAN_MANUAL_BOFF_RECOVERY ioctl call arguments. 5-1025
5-865 MSCAN_LISTEN_ONLY_MODE ioctl call arguments 5-1026
5-866 MSCAN_LOOPBACK_MODE ioctl call arguments 5-1027
5-867 MSCAN_SET_CLOCK_SOURCE ioctl call arguments 5-1028
5-868 MSCAN_SET_PRESCALER ioctl call arguments 5-1029
5-869 MSCAN_SET_RJW ioctl call arguments. 5-1030
5-870 MSCAN_SET_TSEG1 ioctl call arguments. 5-1031
5-871 MSCAN_SET_TSEG2 ioctl call arguments. 5-1032
5-872 MSCAN_SET_SAMPLING ioctl call arguments 5-1033
5-873 MSCAN_SET_ACC_MODE ioctl call arguments. 5-1034
5-874 MSCAN_SET_ACC_MASKR_32_x ioctl call arguments 5-1035
5-875 MSCAN_SET_ACC_MASKR_16_x ioctl call arguments 5-1036
5-876 MSCAN_SET_ACC_MASKR_8_x ioctl call arguments 5-1037
5-877 MSCAN_SET_ACC_IDR_32_x ioctl call arguments 5-1038
5-878 MSCAN_SET_ACC_IDR_16_x ioctl call arguments 5-1039
5-879 MSCAN_SET_ACC_IDR_8_x ioctl call arguments 5-1040
5-880 MSCAN_SLEEP ioctl call arguments 5-1041
5-881 MSCAN_GET_SLEEP_MODE ioctl call arguments 5-1042
5-882 MSCAN_AUTO_WAKEUP ioctl call arguments 5-1043
5-883 MSCAN_TIMESTAMP_TIMER ioctl call arguments 5-1044
5-884 MSCAN_TEST_SYNCH ioctl call arguments. 5-1045
5-885 MSCAN_TEST_RXACT ioctl call arguments. 5-1046
5-886 MSCAN_TEST_RXFRM ioctl call arguments 5-1047
5-887 MSCAN_CLEAR_RXFRM ioctl call arguments. 5-1048
5-888 MSCAN_STOP_IN_WAIT ioctl call arguments 5-1049

5-889 MSCAN_ERINT_ENABLE ioctl call arguments 5-1050
5-890 MSCAN_ERINT_DISABLE ioctl call arguments 5-1051
5-891 MSCAN_ERINT_SET_RSTATE_MODE ioctl call arguments 5-1052
5-892 MSCAN_ERINT_SET_TSTATE_MODE ioctl call arguments 5-1053
5-893 MSCAN_TINT_ENABLE ioctl call arguments 5-1054
5-894 MSCAN_TINT_DISABLE ioctl call arguments 5-1055
5-895 MSCAN_GET_ENABLED_TINT ioctl call arguments 5-1056
5-896 MSCAN_READ_ERINT_FLAGS ioctl call arguments 5-1057
5-897 MSCAN_READ_EINT_FLAGS ioctl call arguments 5-1058
5-898 MSCAN_READ_TINT_FLAGS ioctl call arguments 5-1059
5-899 MSCAN_CLEAR_ERINT_FLAGS ioctl call arguments 5-1060
5-900 MSCAN_CLEAR_EINT_FLAGS ioctl call arguments 5-1061
5-901 MSCAN_CLEAR_RINT_FLAG ioctl call arguments 5-1062
5-902 MSCAN_CLEAR_WINT_FLAG ioctl call arguments 5-1063
5-903 MSCAN_SELECT_TXBUFF ioctl call arguments 5-1064
5-904 MSCAN_SELECT_NEXT_TXBUFF ioctl call arguments 5-1065
5-905 MSCAN_TRANSMIT ioctl call arguments 5-1066
5-906 MSCAN_ABORT_TRANSMIT ioctl call arguments 5-1067
5-907 MSCAN_READ_ABORT_ACK ioctl call arguments 5-1068
5-908 MSCAN_TEST_BUSOFF_HOLD ioctl call arguments 5-1069
5-909 MSCAN_RECOVER_BUSOFF_STATE ioctl call arguments 5-1070
5-910 MSCAN_GET_WINNING_ACC_FILTER ioctl call arguments 5-1071
5-911 MSCAN_GET_RX_ERR_COUNT ioctl call arguments 5-1072
5-912 MSCAN_GET_TX_ERR_COUNT ioctl call arguments 5-1073
5-913 MSCANMB_GET_ID ioctl call arguments 5-1075
5-914 MSCANMB_GET_ID_RAW ioctl call arguments 5-1076
5-915 MSCANMB_GET_LEN ioctl call arguments 5-1077
5-916 MSCANMB_GET_LEN ioctl call arguments 5-1078
5-917 MSCANMB_GET_DATAPTR ioctl call arguments 5-1079
5-918 MSCANMB_GET_TIMESTAMP ioctl call arguments. 5-1080
5-919 MSCANMB_SET_ID ioctl call arguments 5-1081
5-920 MSCANMB_SET_ID_V ioctl call arguments 5-1083
5-921 MSCANMB_SET_ID_RAW ioctl call arguments. 5-1084
5-922 MSCANMB_SET_RTR ioctl call arguments. 5-1085
5-923 MSCANMB_SET_LEN ioctl call arguments. 5-1086
5-924 MSCANMB_SET_TBP ioctl call arguments 5-1087
5-925 RTC Module Base Address 5-1089
5-926 RTC Configuration Items for appconfig.h 5-1090
5-927 RTC Driver Arguments - ioctl 5-1090

5-928 ioctl commands 5-1091
5-929 RTC_INIT ioctl call arguments 5-1093
5-930 RTC_SET_CLOCK_SOURCE_PRESCALER ioctl call arguments 5-1094
5-931 RTC_SET_CLOCK_PRESCALER ioctl call arguments 5-1095
5-932 RTC_CLOCK_SOURCE ioctl call arguments 5-1096
5-933 RTC_SET_INTERRUPT_ENABLE ioctl call arguments 5-1097
5-934 RTC_SET_INTERRUPT_DISABLE ioctl call arguments 5-1098
5-935 RTC_TEST_INTERRUPT_FLAG ioctl call arguments 5-1099
5-936 RTC_CLEAR_INTERRUPT_FLAG ioctl call arguments. 5-1100
5-937 RTC_WRITE_MODULO_REG ioctl call arguments 5-1101
5-938 RTC_READ_MODULO_REG ioctl call arguments 5-1102
5-939 RTC_READ_COUNTER_REG ioctl call arguments 5-1103
6-1 FreeMASTER Driver Interrupt Mode 6-3
6-2 FreeMASTER Communication Configuration Items for appconfig.h 6-5
6-3 TSA Type Constants 6-23

List of Figures

1-1 Software Structure 1-2
2-1 Boot Sequence 2-2
2-2 Interrupt Processing Flow 2-25
2-3 Memory Checking Process 2-47
3-1 Root Directory Structure 3-1
3-2 Sample Applications Directory Structure 3-2
3-3 Src Directory Structure 3-3
3-4 Stationery Directory Structure 3-4
4-1 User Interface 4-3
5-1 Macro Expansion Process 5-3
5-2 IIC Bus Transmission Signals 5-774
5-3 IIC Interrupt Handling 5-818
6-1 FreeMASTER Application Window 6-2
7-1 GCT Usage 7-2
7-2 GCT Main Window 7-3
7-3 Pinout Page Status Icons 7-4
7-4 Pinout Page 7-5
7-5 Register View 7-5
7-6 Warnings View 7-6
7-7 Options dialog 7-7
7-8 The appconfig.h File 7-8

Chapter 1
Introduction

This user's manual is targeted for Freescale 56F8xxx application developers. Its purpose is to describe the development environment, the software modules and the tools for the 56F8xxx and the Application Programming Interface (API). Simply, this manual describes how to use the Freescale DSP56800E_Quick_Start tool to develop software for the Freescale 56F8xxx Digital Signal Controllers (DSC).

1.1 Overview

The DSP56800E_Quick_Start development environment provides fully debugged peripheral drivers, examples and interfaces, that allow programmers to create their own C application code, independent of the core architecture. This environment has been developed to complement the existing development environment for Freescale 56F8xxx embedded processors. It provides a software infrastructure that allows development of efficient, ready to use high level software applications, that are fully portable and reusable between different core architectures. The maximum portability is achieved for devices with comparable on-chip peripheral modules.

This manual contains information specific only to DSP56800E_Quick_Start tool as it applies to the Freescale 56F8xxx software development. Therefore it is required that users of the DSP56800E_Quick_Start tool should be familiar with the 56800E family in general, as described in the ***DSP56800E 16-Bit DSP Core Reference Manual*** (DSP56800ERM/D), ***MC56F8300 Peripheral User Manual*** (MC56F8300UM/D) and the **56F8000 Peripheral Reference Manual** (MC56F8000RM), before continuing. The 56F8xxx devices are supported by a complete set of hardware development boards - evaluation modules (EVMs) and development system cards for fast system development (e.g. Legacy Motor Interface Daughter Card).

Comprehensive information about available tools and documentation can be found on Freescale web pages:

`http://www.freescale.com/`

Freescale DSP56800E_Quick_Start tool is designed for and can be fully integrated with Freescale CodeWarrior development tools. Before starting to explore the full feature set of the DSP56800E_Quick_Start, one should install and become familiar with the CodeWarrior development environment.

All together, the DSP56800E_Quick_Start, the CodeWarrior, and the EVMs create a complete and scalable tool solution for easy, fast and efficient development.

1.1.1 Features

The DSP56800E_Quick_Start environment is composed of the following major components: core-system infrastructure, on-chip drivers with defined API, sample example applications, Graphical Configuration Tool and FreeMASTER software support. This section brings very illustrative information about these components, while the comprehensive description can be found in specially targeted chapters.

1.1.1.1 Core-system Infrastructure

The core-system infrastructure creates the fundamental infrastructure for the 56F8xxx device operation and enables further integration with other components, e.g. on-chip drivers. The basic development support provided includes: setting of the required operation mode, commonly used macro definitions, portable architecture-dependent register declaration, mechanism for static configuration of on-chip peripherals as well as interrupt vectors, and the project templates.

1.1.1.2 On-chip Drivers

The on-chip drivers isolate the hardware-specific functionality into a set of driver commands with defined API. The API standardizes the interface between the software and the hardware, see Figure 1-1. This isolation enables a high degree of portability or architectural and hardware independence for application code. This is mainly valid for devices with similar peripheral modules. The driver code reuses lead for greater efficiency and performance.

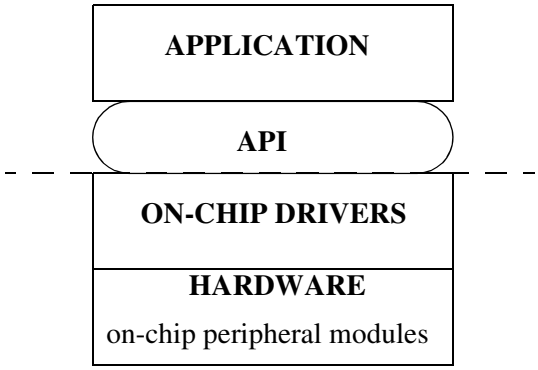


Figure 1-1. Software Structure

1.1.1.3 Sample Applications

The DSP56800E_Quick_Start tool contains many sample applications that demonstrate how to use on-chip drivers and how to implement some user-specific tasks. These sample examples are kept simple and illustrative and their intention is to minimize the learning curve.

1.1.1.4 Graphical Configuration Tool

The Graphical Configuration Tool (GCT) is a graphical user interface (GUI), designed to provide static chip and on-chip peripheral module setting/initialization, including association of the interrupt vectors with user interrupt service routines.

The Graphical Configuration Tool is not required in order to use the DSP56800E_Quick_Start environment, i.e. it is optional. Nevertheless, this tool simplifies the configuration of on-chip peripheral modules and the device itself. It also guides the user by supplying a lot of useful information and hints. It is therefore recommended to use the Graphical Configuration Tool.

1.1.1.5 FreeMASTER Software

The FreeMASTER application is a software tool initially created for developers of Motor Control applications, but it may be extended to any other application development. This tool allows remote control of an application using a user-friendly graphical environment running on a PC. It also provides the ability to view some real-time application variables in both textual and graphical form.

Main features:

- Graphical environment
- Visual Basic Script or Java Script can be used for control of target board
- Easy to understand navigation
- Connection to target board possible over a network, including Internet
- Demo mode with password protection support
- Visualization of real-time data in Scope window
- Acquisition of fast data changes using integrated Recorder
- Value interpretation using custom defined text messages
- Built-in support for standard variable types (integer, floating point, bit fields)
- Several built-in transformations for real type variables
- Automatic variable extraction from CodeWarrior linker output files (MAP, ELF)
- Remote control of application execution

The FreeMASTER tool is not required in order to use the DSP56800E_Quick_Start environment, i.e. it is optional. Nevertheless, FreeMASTER is a versatile tool to be used for multipurpose algorithms and applications. It provides a lot of excellent features, including:

- Real-Time debugging
- Diagnostic tool
- Demonstration tool
- Education tool

The full description can be found in the ***FreeMASTER User Manual*** attached to the FreeMASTER tool.

1.2 Quick Start

This chapter provides the information required to get the DSP56800E_Quick_Start tool installed and running.

1.2.1 Install CodeWarrior Development Tools

CodeWarrior Development Studio 56800/E Hybrid Controllers provides a complete software development environment for Freescale hybrid controllers. CodeWarrior is a windows based Integrated Development Environment (IDE) with highly efficient C compilers.

As previously mentioned, Freescale DSP56800E_Quick_Start tool is designed for and can be integrated with CodeWarrior development tools. With CodeWarrior tools, users can build applications and integrate other software included as part of the DSP56800E_Quick_Start release. Once the software is built, CodeWarrior tools allows users to download executable images into the target platform and run or debug the downloaded code.

The rest of this section describes the general installation process of the CodeWarrior tools. However, it is recommended to use the installation guide attached to the actual version of CodeWarrior, if available.

To start the installation process, perform the following steps:

1. Insert the CodeWarrior CD-ROM into your computer's CD-ROM drive.

If Auto Install is disabled on your computer, click the **Start** button, select **Run**, and type the CD-ROM's drive letter and \Setup.exe in the **Open**: text box. (e.g. **D:\Setup.exe**)

2. Follow the CodeWarrior software installation instructions on your screen.

Note: *After installing CodeWarrior, remember to restart the computer. This restart ensures that newly installed drivers will be available for use.*

3. Register CodeWarrior

- Click the **Start** button, then select **CodeWarrior Registration** from the CodeWarrior group. This runs the MWRegister.exe program.
- Enter your registration number and contact information. If you do not have a registration number, leave the Registration Number text box blank. • Click **OK** and send the resulting text file (e.g. **MWRegistration.tx**t) to license@metrowerks.com. A license key will be sent to you via E-mail.

4. Install the license key.

- Locate the license.dat file in the CodeWarrior installation directory and open this file with any standard text editor, for example, Notepad.
- Copy or type the key starting on a new line at the bottom of the license.dat file. For more detailed instructions see the License_Install.txt file in the Licensing directory.
- Save the license.dat file.

1.2.2 Install DSP56800E_Quick_Start

In order for the DSP56800E_Quick_Start to integrate itself with the development tools, the CodeWarrior tools should be installed prior to the installation of DSP56800E_Quick_Start installation (see previous section). If the DSP56800E_Quick_Start tool is installed while CodeWarrior is not present, users can only browse the installed software package, but will not be able to build, download and run the released code. However, the installation can be simply completed once CodeWarrior is installed, see Section 1.2.2.1.

The installation itself consists of copying the required files to the destination hard drive, checking the presence of CodeWarrior and creating the shortcut under the Start->Programs menu.

Note: Each DSP56800E_Quick_Start release is installed in its own new directory named *DSP56800E_Quick_Start rX.Y* (where *X.Y* denotes the release number). Thus, it enables to maintain the older releases and projects. It gives free choice to select the active release.

To start the installation process, perform the following steps:

1. Execute DSP56800E_Quick_Start_rXY.exe

2. Follow the DSP56800E_Quick_Start software installation instructions on your screen.

To integrate DSP56800E_Quick_Start with CodeWarrior, perform the following steps:

3. Set path to the DSP56800E_Quick_Start source within IDE

   a) Launch CodeWarrior IDE from the Start->Programs->Freescale CodeWarrior menu

   b) Open *IDE Preferences* dialog window using Edit->Preferences...

   c) Select *Source Trees* panel from IDE Preferences Panels-General

   d) Type ***DSP56800E_Quick_Start Source*** to the Name box

   e) Choose *Absolute Path* as a path type

   f) Click *Choose* and locate the DSP56800E_Quick_Start installation directory, e.g. C:\Program Files\Freescale\DSP56800E_Quick_Start r2.4\src

   g) click *Add*

   h) click *OK* to finish

Setting Up a Remote Connection. A remote connection is a type of connection to use for debugging along with any preferences that connection may need. To create a new remote connection for the DSP56800E_Quick_Start:

   a) Launch CodeWarrior IDE from the Start->Programs->Freescale CodeWarrior menu

   b) On the main menu, select Edit -> Preferences

   c) Click *Remote Connections* in the left column

   d) Click the *Add* button - the *New Connection* window appears

   e) In the Name edit box, type in *HW* as the connection name

   f) In the Debugger combo box, select *CCS 56800E Protocol Plug-in*

   g) In the Connection Type combo box, select *CCS Remote Connection*

   h) Leave the check boxes unchecked (this is the default state)

   i) Click the OK button

j) Click again the *Add* button - The *New Connection* window appears

k) In the Name edit box, type in *SIM* as the connection name

l) In the Debugger combo box, select *Sim 56800E Protocol Plug-in*

m) In the Connection Type combo box, select *Simulator*

n) In the Simulation BandWidth combo box, select *Medium* (this is the default state)

o) Click the OK button

1.2.2.1 Supplementary DSP56800E_Quick_Start Installation Steps

This section describes the additional installation steps required if CodeWarrior is installed after the DSP56800E_Quick_Start installation. Note: if this condition does not apply to you, skip this section completely.

Suppose that CodeWarrior and DSP56800E_Quick_Start are now successfully installed. The next step is to copy the content (i.e. the subdirectory) of the DSP56800E_Quick_Start Stationery folder (e.g. ...Freescale\DSP56800E_Quick_Start r2.4\stationery) into the CodeWarrior root folder (e.g. ...\CodeWarrior). This operation “registers” the DSP56800E_Quick_Start project templates for the newly created projects. It is necessary to integrate DSP56800E_Quick_Start with CodeWarrior as the last step. To do this, perform the following steps:

a) Launch CodeWarrior IDE from the Start->Programs->Freescale CodeWarrior menu

b) Open the *IDE Preferences* dialog window using Edit->Preferences...

c) Select the *Source Trees* panel from IDE Preferences Panels-General

d) Type ***DSP56800E_Quick_Start Source*** to the Name box

e) Choose *Absolute Path* as a path type

f) Click *Choose* and locate the DSP56800E_Quick_Start installation directory, e.g. C:\Program Files\Freescale\DSP56800E_Quick_Start r2.4\src

g) click *Add*

h) click *OK* to finish

1.2.2.2 Install Graphical Configuration Tool

The Graphical Configuration Tool is installed together with the whole DSP56800E_Quick_Start environment as a part of the *Typical* installation. The graphical configuration tool can also be installed as a selectable component within the *Custom* installation.

The Graphical Configuration Tool is able to work as stand-alone, but integration with the CodeWarrior IDE increases markedly the efficiency of this tool. This integration is based on the IDE user-configurable menus and its interface for external plug-ins.

To integrate the Graphical Configuration Tool with CodeWarrior IDE, perform the following steps:

a) Launch CodeWarrior IDE from the Start->Programs->Freescale CodeWarrior menu

b) Open the *Commands and Key Bindings* dialog window using Edit->Commands and Key Bindings

c) Unroll the desired menu command group (e.g. *Project*) from the command tree on the left side of the dialog box.

d) Click on any item of unrolled tree

e) Click on button *New Command*

f) Type *Co&nfiguration Tool* to the Name box (char '&' in front of char 'n' enables selecting assigned menu item using Alt-n)

g) Click on check box *Appears in Menus* and tick it.

h) Click on button on the right side of the *Execute* edit box and browse for gct56F800.exe

i) Click on button on the right side of the *Arguments* edit box and select item Project File directory from popup menu

j) If you want to assign key binding, click on button *New Binding* and press chosen key combination, e.g. Ctrl+F12

k) Press button Save and close the dialog box

Now you should be able to execute the Graphical Configuration Tool from the CodeWarrior IDE menu or by pressing the chosen key shortcut. Note that the DSP56800E_Quick_Start project should be open in the IDE to quickly execute the Graphical Configuration Tool.

1.2.2.3 Install FreeMASTER (PC Master Software)

1.2.2.3.1 System Requirements

The FreeMASTER application can run on any computer with Microsoft Windows 98 or later operating system. Before installing, the Internet Explorer 4.0 or higher (5.5 is recommended) should be installed.

The following requirements result from those for the Internet Explorer 4.0 application:

Computer: 486DX/66 MHz or higher processor

Operating system: Microsoft Windows XP, Windows 2000, Windows NT4 with SP6, Windows 98

Required software: Internet Explorer 4.0 or higher installed. For selected features (e.g. regular expression-based parsing), Internet Explorer 5.5 or higher is required.

Hard drive space: 8 MB

Other hardware requirements: Mouse, serial RS-232 port for local control, network access for remote control

1.2.2.3.2 Target Development Board Requirements

To enable the FreeMASTER connection to the target board application, follow the instructions provided with the embedded-side development tool. The recommended and fastest way to start using FreeMASTER is by trying the sample application. Note that the sample application name may still refer the “PC Master” software, which is the previous name of the FreeMASTER tool. FreeMASTER is fully backward compatible with PC Master.

FreeMASTER software relies on the following items to be provided by the target development board:

Interface: Serial communication port or the JTAG port (available on all Freescale EVM boards).

Data RAM Memory: Approximately 160 words of data memory plus the size of the recorder buffer is needed for the full configuration. Optionally, some features can be disabled to reduce required data memory size.

Program Flash Memory: Required size is approximately 2K words for the full configuration. Optionally, some features can be removed to reduce required program memory size

1.2.2.3.3 Enabling FreeMASTER on Target Application

To enable the FreeMASTER operation on the target board application, see description and an example in Chapter 6, “FreeMASTER Driver.” .

1.2.2.3.4 How to Install

The FreeMASTER application is an optional part of the DSP56800E_Quick_Start environment and must be installed separately, e.g. running the **fmaster13-1.exe**.

1.2.3 Install MC56F8xxxEVM Hardware

The DSP56800E_Quick_Start for 56F8xxx has been designed and tested with the MC56F83xxEVM or MC56F8xxxDEMO target hardware. If the user wants to quickly exercise software applications included with DSP56800E_Quick_Start, MC56F8xxxEVM/DEMO hardware must be installed.

The MC56F8xxxEVM/DEMO installation information is provided with CodeWarrior installation and can be found in the following document **(located in the CodeWarrior installation directory)**:

<...>\CodeWarrior\CodeWarrior Manuals\PDF\Targeting_DSP56800E.pdf

It is recommended that all DSP56800E_Quick_Start users read through this document, before proceeding with software development.

1.2.4 Build and Run Sample Application

Once the DSP56800E_Quick_Start tool is installed, the user can build and run any of the released demo applications for the MC56F83xxEVM / MC56F8xxxDEMO by opening and building the project, using the CodeWarrior development environment. We will use *pwm_demo.mcp* as an example:

Step 1: Launch CodeWarrior IDE from the Start->Programs->Freescale CodeWarrior menu.

Step 2: Using File->Open command, open the *pwm_demo.mcp* project by selecting, e.g. ..\DSP56800E_Quick_Start r2.4\sample_applications\MC56F8346EVM\pwm_demo directory.

Note: *Select the corresponding directory according to your EVM board, for the MC56F8346EVM, open the ..\DSP56800E_Quick_Start r2.4\sample_applications\MC56F8346EVM\pwm_demo directory.*

Step 3: Execute the application in the *Debug* mode by pressing the *F5* key or choose the *Debug* command from the *Project* menu.

Step 4: Run the application by pressing the green arrow (Run) in the debug window or choose the *Run* command from the *Project* menu.

At this point, the application is running - the LEDs associated to the PWM outputs are flashing now and the green LED is blinking periodically.

The subsequent chapters describe how to create a new application, how to use interrupts, how to use on-chip drivers and other information required to successfully create a new application.

Chapter 2
Core System Infrastructure

The Core System Infrastructure is one of the three main blocks that compose the DSP56800E_Quick_Start tool (see Section 1.1.1 where the partitioning is described). Its purpose is to provide the fundamental infrastructure for the 56800E device operation (e.g. sets the operation mode, the interrupt handling, the initialization of the global variables, CodeWarrior Compiler options). It also provides some additional support (commonly used macros, data types) and enables further integration with On-chip Drivers.

2.1 Boot Sequence

The Core System Infrastructure provides the fundamental code which is executed before the user's main function. This code provides basic settings needed to initialize the chip, settings required by the CodeWarrior Compiler, initialization of global variables. Finally it passes control to the user's application code (the *main* function).

Note: This chapter describes the boot process of the MC56F83xx family of microcontrollers which contain the Boot Flash memory. The boot process of the devices without the dedicated Boot Flash memory (MC56F80xx) is relatively simpler as the vector table need not to be relocated. Except this difference, the other startup steps are common for all 56800E devices.

For the MC56F83xx devices, the post-reset execution flow may be briefly described as follows (also see Figure 2-1):

1. After processor reset, the execution starts at the Hardware Reset vector in program memory, where the DSP56800E_Quick_Start tool places its jump to the *Start()* assembly routine
   - — For the EXTBOOT=1 and EMI_MODE=0 processor configuration (processor external pins are sampled during reset), the reset vector is located at address 0x0000 and the jump is supplied directly from the first entry of vector table located in the *interrupt_vectors* section in *vectors.c* file.
   - — For another configuration, the reset vector is located at address 0x20000. A jump to the *Start()* routine is compiled on this address (in the *boot_jump* section in *vectors.c* file) while keeping the full vector table on address 0x0000.
2. If the chip reset is generated by the watchdog module (COP), the same rules as in the previous point apply, except that the second entry of the vector table is used. Again the COP Reset vector is supplied from either the full vector table at address 0x0000 (*interrupt_vectors* section) or the reset jumps table at address 0x20000 (*boot_jump* section). The default value of COP Reset vector is *Start()*, so the standard power-up code is processed. The user is able to redefine the COP Reset service routine same way the other interrupt vectors are installed (see Section 2.5.2 on page 2-27 for more details).

3. *Start()* assembly routine (in *startup.c* file)
4. *userPreMain()* function (in *appconfig.c* file)
5. user's *main()* function (in default project it is located in *main.c*)
6. *userPostMain()* function (in *appconfig.c* file)

The following subsections provide a detailed description of all initialization performed before user's *main()* function is called.

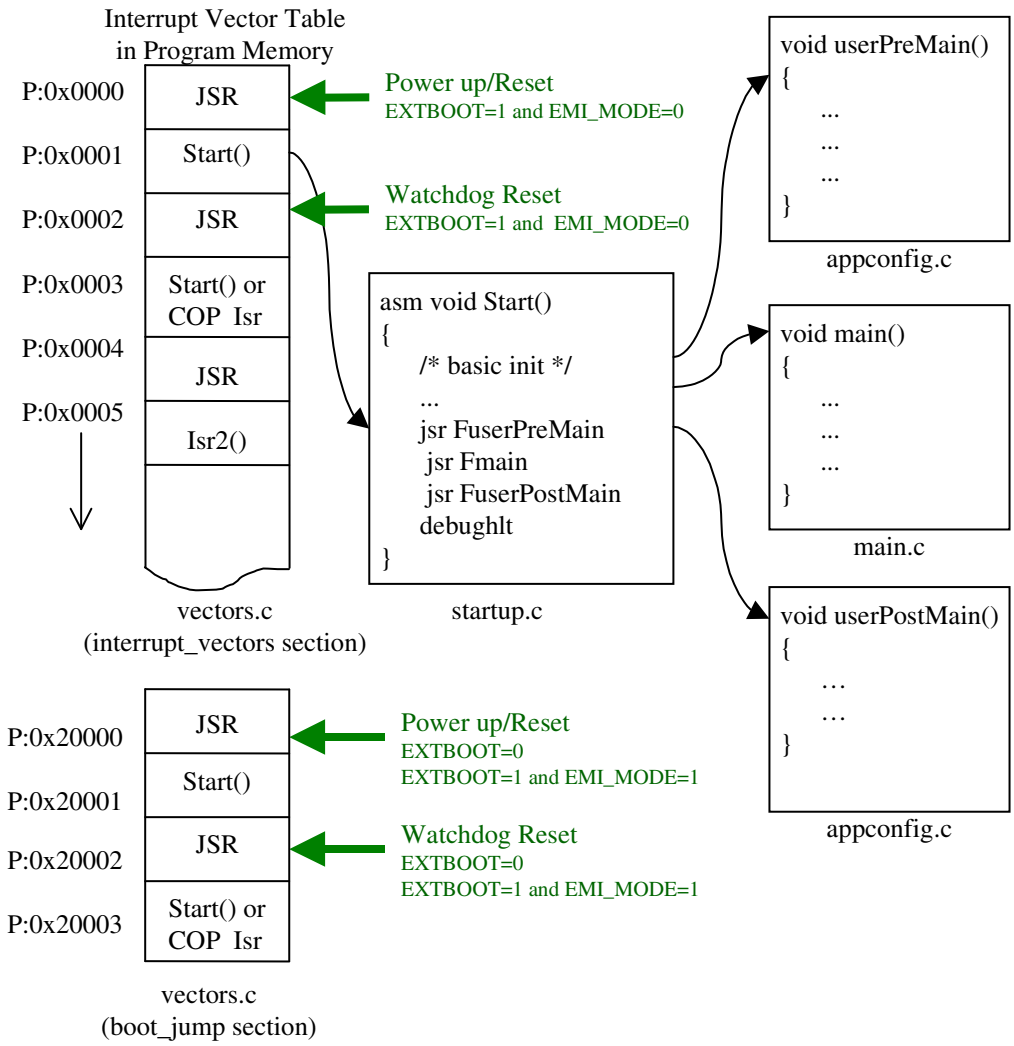


Figure 2-1. Boot Sequence

2.1.1 Power-up/Reset

The 56800E core specifies two reset vectors: Hardware Reset and COP Watchdog Reset. These reset vectors are located at first two locations of interrupt vector table at address 0x0000 or 0x20000 depending on the configuration of the EXTBOOT and EMI_MODE pins. These vectors identify the address of the program code where the program control is passed to on reset. In applications developed with the DSP56800E_Quick_Start tool, the default entry point is the *Start()* assembly routine in the *startup.c* file.

In the DSP56800E_Quick_Start tool, the vector table (*vectors.c*) is always linked at address 0x0000 for any processor configuration. To assure the startup code gets called after reset even for the configurations where 0x20000 is the default vectors base, two jumps to the *Start()* are linked at these addresses from the *boot_jump* section of the *vectors.c* file for both reset vectors. The startup code then configures the interrupt controller to use the address 0x0000 as the vector table base address.

2.1.2 *Start()* - entry point

The entry point of all projects developed with the DSP56800E_Quick_Start tool is the *Start()* assembly routine located in the *startup.c* file in {project}\SystemConfig directory. This routine performs the following initialization:

- configures the interrupt controller to use the address 0x0000 as the vector base address
- sets the OMR register according to the settings in global application configuration file (*appconfig.h*)
- initializes the On-chip Clock Synthesis (OCCS) module, sets the PLL (by values from *appconfig.h*) and waits while the generated clock is stable
- sets the External Memory Interface unit (SEMI) to generate the proper chip select signals and the wait states for the external memories according to the *appconfig.h* file
- [optionally] runs the internal memory tests with halting the processor if memory module is not usable
- initializes the stack pointer (SP) to the address after any data segments
- clears the *.bss* segment which holds the uninitialized global and static C variables
- copies the initial values from Flash memory to initialized global C variables (*.data* segment). Either xFlash or pFlash memory can be chosen to hold the initialization data.
- clears and initializes variables in the *fardata.bss* and *fardata.data* segments
- clears and initializes variables in the *.bss.pmem* and *.bss.data* segment (program RAM-based variables)
- initializes the program RAM-based code of the *pramcode* section.

When all the initialization is done, the functions *userPreMain()*, *main()*, *userPostMain()* are called.

2.1.3 userPreMain()

The *userPreMain()* function is called before the main application code in the *main()* function. The user can add any additional initialization code here. The function is located in the *appconfig.c* file.

2.1.4 *main()* the User's Application Code

The *main()* function is called after all the code described above is executed (i.e. the processor is initialized and the user's pre-main code is executed). It is the place where the user writes the application code. By default the function is located in the *main.c* file, but the file can be renamed by the user.

2.1.5 userPostMain()

The *userPostMain()* function is called after the main application code is finished. The user can add any additional code he/she wishes. By default the processor is halted by *debughlt* instruction here. The function is located in the *appconfig.c* file.

2.2 Data Types

The DSP56800E_Quick_Start tool defines some basic data types to support code portability between different hardware architectures and tools. These basic data types, which are defined in the C header file *types.h,* support International Telecommunication Union (ITU) generic word types, integer, fractional, and complex data types. This is used throughout the interface definitions for the On-Chip Drivers. Note that in some development environments these data type definitions are located in the *prototype.h* file.

1. Generic word types
   - Word8 - to represent 8-bit signed character variable/value
   - UWord8 - to represent 16-bit unsigned character variable/value
   - Word16 - to represent 16-bit signed variable/value
   - UWord16 - to represent 16-bit unsigned variable/value
   - Word32 - to represent 32-bit signed variable/value
   - UWord32 - to represent 32-bit unsigned variable/value
2. Integer types
   - Int8 - to represent 8-bit signed character variable/value
   - UInt8 - to represent 8-bit unsigned character variable/value
   - Int16 - to represent 16-bit signed variable/value
   - UInt16 - to represent 16-bit unsigned variable/value
   - Int32 - to represent 32-bit signed variable/value
   - UInt32 - to represent 32-bit unsigned variable/value
3. Fractional types
   - Frac16 - to represent 16-bit signed variable/value
   - Frac32 - to represent 32-bit signed variable/value
   - CFrac16 - to represent 16-bit complex numbers
   - CFrac32 - to represent 32-bit complex numbers
4. Miscellaneous types
   - bool - to represent boolean variable (true/false)
5. Constants
   - true - represents true value
   - false - represents false value
   - NULL - represents null pointer
   - MAX_32 - maximum 32-bit signed (Word32) value
   - MIN_32 - minimum 32-bit signed (Word32) value
   - MAX_16 - maximum 16-bit signed (Word16) value
   - MIN_16 - minimum 16-bit signed (Word16) value

2.3 *ArchIO* Peripheral Register Structures

The global symbol *ArchIO* provides a C interface (structure type) to all peripheral and core registers mapped in data memory. All registers are accessed via this structure so there is no need to know and specify the concrete addresses of the registers to write or read. This mechanism increases code readability and portability and simplifies access to registers. The *ArchIO* is declared in the C header file *arch.h.*

The *ArchIO* is of type *arch_sIO, which is* the structure type composed from another structures, one for each peripheral module.

There are two possible approaches how to define and use the ArchIO structure:

- define ArchIO as the direct (numeric) address of memory-mapped peripheral registers casted to the proper structure type.
- define ArchIO as the *extern* variable while defining its address by a directive in linker command file.

The second approach is used in the DSP56800E_Quick_Start tool implementation by default.

Example 2-1. Using the ArchIO structure

```
UWord16 RegValue;

RegValue = ArchIO.TimerD.Channel0.HoldReg;
ArchIO.TimerD.Channel0.CompareReg1 = 0x8000;
```

The Code Example 2-1 reads the timer/counter D0 Hold Register (HOLD) and writes to the timer/counter D0 Compare Register 1 (CMP1).

Example 2-2. Using the ArchIO structure

```
UWord16 RegValue;

RegValue = periphMemRead(&ArchIO.TimerD.Channel0.HoldReg);
periphMemWrite(0x8000, &ArchIO.TimerD.Channel0.CompareReg1);
```

Code Example 2-2 shows the same operation using the *periphMemRead* and *periphMemWrite* macros described later in Section 2.4.2:

2.4 Core System's Routines and Macros

This section describes routines, macros and intrinsic function redefinition provided by the Core System Infrastructure.

2.4.1 Architecture dependent routines

This section describes architecture dependent routines and macros which provide interface to the 56800E core architecture. It encapsulates the unique features of the 56800E architecture into the abstract APIs. All routines are defined in the *arch.h* header file.

2.4.1.1 archEnableInt - enable interrupts

Call(s):

```
void archEnableInt(void);
```

Arguments: None.

Description: The *archEnableInt* macro enables all interrupts by clearing bits I1 (Bit 9) and I0 (Bit 8) in the Status Register (SR).

Example 2-3. archEnableInt macro usage

```
archEnableInt();
```

2.4.1.2 archEnableIntLvl123 - enable interrupt levels 1, 2 and 3

Call(s):

```
void archEnableIntLvl123(void);
```

Arguments: None.

Description: The *archEnableInt*Lvl123 macro enables interrupts at levels 1, 2 and 3 while masking the interrupts at level 0. It is accomplished by clearing bit I1 (Bit 9) and setting bit I0 (Bit 8) in the Status Register (SR).

Example 2-4. archEnableIntLvl123 macro usage

```
archEnableIntLvl123();
```

2.4.1.3 archEnableIntLvl23 - enable interrupts levels 2 and 3

Call(s):

```
void archEnableIntLvl23(void);
```

Arguments: None.

Description: The *archEnableIntLvl23* macro enables interrupts at levels 2 and 3 while masking interrupts at levels 0 and 1. It is accomplished by setting bit I1 (Bit 9) and clearing I0 (Bit 8) in the Status Register (SR).

Example 2-5. archEnableIntLvl23 macro usage

```
archEnableIntLvl23();
```

2.4.1.4 archDisableInt - disable interrupts

Call(s):

```
void archDisableInt(void);
```

Arguments: None.

Description: The *archDisableInt* macro disables all maskable interrupts by setting bits I1 and I0 (Bits 9 - 8) in the Status Register (SR).

Example 2-6. archDisableInt macro usage

```
archDisableInt();
```

2.4.1.5 archResetLimitBit - reset limit bit

Call(s):

```
void archResetLimitBit(void);
```

Arguments: None.

Description: The *archResetLimitBit* macro resets limit bit (L) - Bit 6 in the Status Register (SR).

Example 2-7. archResetLimitBit macro usage

```
archResetLimitBit();
```

2.4.1.6 archSetNoSat - set no saturation mode

Call(s):

```
void archSetNoSat(void);
```

Arguments: None.

Description: The *archSetNoSat* macro disables the saturation mode. This macro clears the saturation (SA) bit - Bit 4 in the Operating Mode Register (OMR).

Example 2-8. archSetNoSat macro usage

```
archResetLimitBit();
```

2.4.1.7 archSetSat32 - set saturation mode

Call(s):

```
void archSetSat32(void);
```

Arguments: None.

Description: The *archSetSat32* macro sets the saturation mode. This macro sets the saturation (SA) bit - Bit 4 in the Operating Mode Register (OMR).

Example 2-9. archSetSat32 macro usage

```
archSetSat32();
```

2.4.1.8 archSet2CompRound - set two's complement rounding mode

Call(s):

```
void archSet2CompRound(void);
```

Arguments: None.

Description: The *archSet2CompRound* macro sets the two's complement rounding mode. This macro sets the rounding (R) bit - Bit 5 in the Operating Mode Register (OMR).

Example 2-10. archSet2CompRound macro usage

```
archSet2CompRound();
```

2.4.1.9 archSetConvRound - set convergent rounding mode

Call(s):

```
void archSetConvRound(void);
```

Arguments: None.

Description: The *archSetConvRound* macro sets the convergent rounding mode. This macro clears the rounding (R) bit - Bit 5 in the Operating Mode Register (OMR).

Example 2-11. archSetConvRound macro usage

```
archSetConvRound();
```

2.4.1.10 archStop - stop processing state

Call(s):

```
void archStop(void);
```

Arguments: None.

Description: The *archStop* macro places the processor into the stop processing state by executing a *stop* instruction.

Example 2-12. archStop macro usage

```
archStop();
```

2.4.1.11 archTrap - initiate a software interrupt

Call(s):

```
void archTrap(void);
```

Arguments: None.

Description: The *archTrap* macro initiates a software interrupt by executing a *swi* instruction.

Example 2-13. archTrap macro usage

```
archTrap();
```

2.4.1.12 archWait - wait processing state

Call(s):

```
void archWait(void);
```

Arguments: None.

Description: The *archWait* macro places the processor into the wait processing state by executing a *wait* instruction.

Example 2-14. archWait macro usage

```
archWait();
```

2.4.1.13 archGetLimitBit - get limit bit

Call(s):

```
Word16 archGetLimitBit(void);
```

Arguments: None.

Description: The *archGetLimitBit* inline function returns the status of the limit bit (L) - Bit 6 in the Status Register (SR).

Returns: The returned value is masked value of the L-bit in SR. It is either 0 - limit bit is cleared or non-zero (0x40) - limit bit is set.

Example 2-15. archGetLimitBit function usage

```
if(archGetLimitBit())
{
      ...
}
```

2.4.1.14 archGetSetSaturationMode - get and set saturation mode

Call(s):

```
Word16 archGetSetSaturationMode(bool bSatMode);
```

Arguments:

Table 2-1. archGetSetSaturationMode arguments

bSatMode	in	State of the saturation mode to be set. false - set no saturation mode true - set saturation mode

Description: The *archGetSetSaturationMode* inline function sets the saturation mode to a user specified value. The function manipulates with the saturation (SA) bit - Bit 4 in the Operating Mode Register (OMR).

Returns: Saturation mode prior to the new state (the return value is masked SA-bit from the previous OMR value).

Example 2-16. archGetSetSaturationMode function usage

```
Word16 bSatMode;
bSatMode = archGetSetSaturationMode(true);
```

2.4.1.15 archDelay - delay

Call(s):

```
void archDelay(UWord16 Ticks);
```

Arguments:

Table 2-2. archDelay arguments

Ticks	in	Number of CPU cycles to delay (0 to 0xFFFF)

Description: The *archDelay* inline function delays the program execution by the specified number of CPU cycles.

Returns: None.

Special Issues: The delay corresponds just roughly to the number of CPU cycles.

Example 2-17. archDelay function usage

```
archDelay(1000);
```

2.4.2 Macros for peripheral memory access

This section describes macros for peripheral memory access. The macros are used to read, write, set, clear, change the memory mapped on-chip peripherals. Using these macros offers a greater portability than simply referencing on-chip peripherals with direct memory accesses. All macros are defined in the *periph.h* header file.

Required Header File(s):

```
#include "types.h"
#include "periph.h"
```

2.4.2.1 periphMemRead - memory read

Call(s):

```
UWord16 periphMemRead(UWord16 *pAddr);
```

Arguments:

Table 2-3. *periphMemRead* arguments

pAddr	in	The memory address from which to read a 16-bit word.

Description: The *periphMemRead* macro reads a 16-bit word from the memory location addressed by parameter pAddr.

Example 2-18. periphMemRead macro usage

```
UWord16 RegValue;

RegValue = periphMemRead(&ArchIO.TimerD.ch0.hold);
```

This code reads the content of the timer/counter D0 Hold Register (HOLD).

2.4.2.2 periphMemWrite - memory write

Call(s):

```
UWord16 periphMemWrite(UWord16 Data, UWord16 *pAddr);
```

Arguments:

Table 2-4. *periphMemWrite* arguments

Data	in	The 16-bit data to write to the memory.
pAddr	in	The memory address to which to write a 16-bit word.

Description: The *periphMemWrite* macro writes a 16-bit word (parameter Data) to the memory addressed by parameter pAddr.

Example 2-19. periphMemWrite macro usage

```
periphMemWrite(0x1234, (UWord16 *) 0x0D60);
periphMemWrite(0xABCD, &ArchIO.TimerD.ch0.cmp1);
```

This code writes 0x1234 to the memory location at address 0x0D60 and value 0xABCD into the timer/counter D0 Compare Register 1.

2.4.2.3 periphBitSet - set selected bits

Call(s):

```
void periphBitSet(UWord16 Mask, UWord16 *pAddr);
```

Arguments:

Table 2-5. *periphBitSet* arguments

Mask	in	Bit mask.
pAddr	in	The memory address.

Description: The *periphBitSet* macro sets the selected bits in a memory location addressed by parameter pAddr.

Example 2-20. periphBitSet macro usage

```
periphBitSet(0xC000, &ArchIO.TimerD.ch0.scr);
```

This code sets bits 15 and 14 in the timer/counter D0 Status and Control Register (SCR).

2.4.2.4 periphMemInvBitSet - invert memory content and set selected bits

Call(s):

```
void periphMemInvBitSet(UWord16 Mask, UWord16 *pAddr);
```

Arguments:

Table 2-6. *periphMemInvBitSet* arguments

Mask	in	Bit mask.
pAddr	in	The memory address.

Description: The *periphMemInvBitSet* macro reads the memory content, inverts its value and sets the selected bits in a memory location addressed by parameter pAddr.

Note, that this macro can be used in some special purposes, e.g. for clearing the pending flags.

Example 2-21. periphMemInvBitSet macro usage

```
periphMemInvBitSet(0x0004, &ArchIO.Sim.rststs);
```

This code clears the Power On Reset flag in the RSTSTS register.

2.4.2.5 periphBitClear - clear selected bits

Call(s):

```
void periphBitClear(UWord16 Mask, UWord16 *pAddr);
```

Arguments:

Table 2-7. *periphBitClear* arguments

Mask	in	Bit mask.
pAddr	in	The memory address.

Description: The *periphBitClear* macro clears the selected bits in a memory location addressed by parameter pAddr.

Example 2-22. periphBitClear macro usage

```
periphBitClear(0xC000, &ArchIO.TimerD.ch0.scr);
```

This code clears bits 15 and 14 in the timer/counter D0 Status and Control Register (SCR).

2.4.2.6 periphBitGrpSR - set bit group to given value

Call(s):

```
void periphBitGrpSR(UWord16 GroupMask, UWord16 Mask,
                    UWord16 *pAddr);
```

Arguments:

Table 2-8. *periphBitSet* arguments

GroupMask	in	Group mask
Mask	in	“ones” bit mask.
pAddr	in	The memory address.

Description: The *periphBitGrpSR* macro sets the bit group to a given value in a memory location addressed by parameter pAddr. All bits specified by GroupMask are affected. These bits are either set if the corresponding bits in Mask value are also set or they are cleared if the corresponding bits in Mask value are cleared.

The “SR” variant uses two non-interruptible instructions *bfset* and *bfclr* to accomplish the requested operation. The *bfset* first sets the “one” bits in the destination location, and *bfclr* then clears the “zero” bits there.

Caution: This macro is the optimal way how to set the specified group of bits to given value. However, it must be kept in mind that during the short time between these two bit operations, the target memory location goes through the third state where the bit group might contain invalid value (“ones” already set but “zeroes” not yet cleared).

Example 2-23. periphBitGrpSR macro usage

```
periphBitGrpSR(0x007f, 10, &ArchIO.Pll.plldb);
```

This code sets the lower 7 bits of PLL Divide-By register to the value 10. Other bits in the register are not affected.

2.4.2.7 periphBitGrpRS - set bit group to given value

Call(s):

```
void periphBitGrpRS(UWord16 GroupMask, UWord16 Mask,
                    UWord16 *pAddr);
```

Arguments:

Table 2-9. *periphBitSet* arguments

GroupMask	in	Group mask

Table 2-9. *periphBitSet* arguments

Mask	in	“ones” bit mask.
pAddr	in	The memory address.

Description: The *periphBitGrpRS* macro sets the bit group to a given value in a memory location addressed by parameter pAddr. All bits specified by GroupMask are affected. The bits are either set if the corresponding bits in Mask value are also set or they are cleared if the corresponding bits in Mask value are cleared.

The “RS” variant uses two non-interruptible instructions *bfclr* and *bfset* to accomplish the requested operation. The *bfclr* first clears the “zero” bits in the destination location, and *bfset* then sets the “one” bits there.

Caution: This macro is the optimal way how to set the specified group of bits to given value. However, it must be kept in mind that during the short time between these two bit operations, the target memory location goes through the third state where the bit group might contain invalid value (“zeroes” already cleared but “ones” not yet set).

Example 2-24. periphBitGrpRS macro usage

```
periphBitGrpRS(0x007f, 10, &ArchIO.Pll.plldb);
```

This code sets the lower 7 bits of PLL Divide-By register to the value 10. Other bits in the register are not affected.

2.4.2.8 periphBitGrpZS - set bit group to given value

Call(s):

```
void periphBitGrpZS(UWord16 GroupMask, UWord16 Mask,
                    UWord16 *pAddr);
```

Arguments:

Table 2-10. *periphBitSet* arguments

GroupMask	in	Group mask
Mask	in	“ones” bit mask.
pAddr	in	The memory address.

Description: The *periphBitGrpZS* macro sets the bit group to a given value in a memory location addressed by parameter pAddr. All bits specified by GroupMask are affected. The bits are either set if the corresponding bits in Mask value are also set or they are cleared if the corresponding bits in Mask value are cleared.

The “ZS” variant uses two non-interruptible instructions *bfclr* and *bfset* to accomplish the requested operation. The *bfclr* first clears all bits in GroupMask and *bfset* then sets the “one” bits there.

Caution: This macro is the optimal way how to set the specified group of bits to given value. However, it must be kept in mind that during the short time between these two bit operations, the target memory location goes through the third state where the bit group contains zeroes.

Example 2-25. periphBitGrpZS macro usage

```
periphBitGrpZS(0x007f, 10, &ArchIO.Pll.plldb);
```

This code sets the lower 7 bits of PLL Divide-By register to the value 10. Other bits in the register are not affected.

2.4.2.9 periphBitGrpSet - set bit group to given value

Call(s):

```
void periphBitGrpSet(UWord16 GroupMask, UWord16 Mask,
                     UWord16 *pAddr);
```

Arguments:

Table 2-11. *periphBitSet* arguments

GroupMask	in	Group mask
Mask	in	“ones” bit mask.
pAddr	in	The memory address.

Description: The *periphBitGrpSet* macro sets the bit group to a given value in a memory location addressed by parameter pAddr. All bits specified by GroupMask are affected. The bits are either set if the corresponding bits in Mask value are also set or they are cleared if the corresponding bits in Mask value are cleared.

This variant uses the accumulator and read-modify-write instructions to accomplish the requested operation. The memory location is first read to accumulator, the *bfclr* and *bfset* instructions are performed on accumulator and the result value is then written back to memory location.

Caution: It might seem this macro is the “proper” way how to set the group of bits to certain value as there are no intermediate invalid values written in the target memory location. However, it is quite dangerous to use this macro when interrupts may occur between the read and write operations. If the interrupt service routine would write the other portion of the target memory location, the written value could be overwritten back with its previous state by the write accumulator operation of periphBitGrpSet.

Example 2-26. periphBitGrpSet macro usage

```
periphBitGrpSet(0x007f, 10, &ArchIO.Pll.plldb);
```

This code sets the lower 7 bits of PLL Divide-By register to the value 10. Other bits in the register are not affected (but see “Caution” above).

2.4.2.10 periphSafeAckByOne - clear (acknowledge) bit flags which are active-high and are cleared by write-one

Call(s):

```
void periphSafeAckByOne(UWord16 GroupMask, UWord16 Mask,
                        UWord16 *pAddr);
```

Arguments:

Table 2-12. *periphSafeAckByOne* arguments

GroupMask	in	Group mask
Mask	in	“ones” bit mask.
pAddr	in	The memory address.

Description: The *periphSafeAckByOne* macro clears (acknowledges) bit flags which are active-high and are cleared by write-one in a peripheral memory location addressed by parameter pAddr. The GroupMask specifies all flags which might be affected by clearing procedure. The Mask value specifies flag/flags to be cleared.

Caution: TBD

Example 2-27. periphSafeAckByOne macro usage

```
periphSafeAckByOne(0x8000 | 0x0100 | 0x0010, 0x0100,
                        &ArchIO.Decoder0.deccr);
```

This code clears the Index Pulse Interrupt Request flag in the Decoder Control Register.

2.4.2.11 periphBitChange - change selected bits

Call(s):

```
void periphBitChange(UWord16 Mask, UWord16 *pAddr);
```

Arguments:

Table 2-13. *periphBitChange* arguments

Mask	in	Bit mask.
pAddr	in	The memory address.

Description: The *periphBitChange* macro complements the selected bits in a memory location addressed by parameter pAddr.

Example 2-28. periphBitChange macro usage

```
periphBitChange(0xC000, &ArchIO.PortB.dr);
```

This code complements bits 15 and 14 in the Port B Data Register (DR).

2.4.2.12 periphBitTest - test selected bits

Call(s):

```
UWord16 periphBitTest(UWord16 Mask, UWord16 *pAddr);
```

Arguments:

Table 2-14. *periphBitTest* arguments

Mask	in	Bit mask.
pAddr	in	The memory address.

Description: The *periphBitTest* macro tests the selected bits if they are set in a memory location addressed by parameter pAddr.

Example 2-29. periphBitTest macro usage

```
if (periphBitTest(0x8000, &ArchIO.TimerD.ch0.scr))
{
      periphBitClear(0x8000, &ArchIO.TimerD.ch0.scr);
};
```

This code checks if Timer Compare Flag (Bit 15) in the timer/counter D0 Status and Control Register (SCR) is set.

2.4.3 Miscellaneous Routines

This section describes some additional routines provided by the DSP56800E_Quick_Start tool.

2.4.3.1 impyuu - integer multiply unsigned 16b x unsigned 16b

Call(s):

```
UWord32 impyuu(UWord16 unsigA, UWord16 unsigB);
```

Arguments:

Table 2-15. impyuu arguments

unsigA	in	first argument
unsigB	in	second argument

Description: The *impyuu* inline function multiplies a 16-bit unsigned integer with a 16-bit unsigned integer and returns the 32-bit unsigned integer result.

Returns: result of multiplication unsigA · unsigB

Example 2-30. impyuu function usage

```
UWord16 var1 = 65535U;
UWord16 var2 = 65535U;
UWord32 result;

result = impyuu(var1, var2); /* returns 4294836225 */
```

This code multiplies variables *var1* and *var2* and returns the result in *result* variable.

2.4.3.2 impysu - integer multiply signed 16b x unsigned 16b

Call(s):

```
Word32 impysu(Word16 sig, UWord16 unsig);
```

Arguments:

Table 2-16. impysu arguments

sig	in	first argument (signed)
unsig	in	second argument (unsigned)

Description: The *impysu* function multiplies 16-bit signed integer and 16-bit unsigned integer as an and returns the 32-bit signed integer result.

Returns: result of multiplication sig · unsig

Example 2-31. impysu function usage

```
Word16 var1 = -32768;
UWord16 var2 = 65535U;
Word32 result;

result = impysu(var1, var2); /* returns -2147450880 */
```

This code multiplies variables *var1* and *var2* and returns the result in *result* variable.

2.4.3.3 shl2 - optimized version of *shl* intrinsic function

Call(s):

```
Word16 shl2(Word16 num, UWord16 shifts);
```

Arguments:

Table 2-17. shl2 arguments

num	in	parameter to be shifted
shifts	in	number of shifts

Description: The *shl2* function performs a multi-bit arithmetic shift of the first parameter to the left by the amount specified in the second parameter. The result is returned as a 16-bit integer. This function is the optimized version of the *shl* intrinsic function (see CodeWarrior Help for more information on *shl*).

Returns: *num* parameter shifted *shifts* times to the left

Example 2-32. shl2 function usage

```
Word16 var1 = 1;
UWord16 var2 = 15;
Word16 result;

result = shl2(var1, var2); /* returns 0x8000 */
```

This code shifts *var1* variable *var2* times to the left and returns the result in *result* variable.

2.4.3.4 shr2 - optimized version of *shr* intrinsic function

Call(s):

```
Word16 shr2(Word16 num, UWord16 shifts);
```

Arguments:

Table 2-18. shr2 arguments

num	in	parameter to be shifted
shifts	in	number of shifts

Description: The *shr2* function performs a multi-bit arithmetic shift of the first parameter to the right by the amount specified in the second parameter. The result is returned as a 16-bit integer. This function is the optimized version of the *shr* intrinsic function (see CodeWarrior Help for more information on *shr*).

Returns: *num* parameter shifted *shifts* times to the right

Example 2-33. shr2 function usage

```
Word16 var1 = 16;
Word16 result;

result = shr2(var1, 3); /* returns 0x0002 */
```

This code shifts *var1* variable three times to the right and returns the result in the *result* variable.

2.4.4 Intrinsic Functions

The DSP56800E_Quick_Start tool exploits the system intrinsic functions defined in *intrinsics_56800E.h* header file distributed with the CodeWarrior Development Studio 56800/E Hybrid Controllers.

To preserve compatibility with the DSP56800_Quick_Start tool, the *intrinsics_56800E.h* is included in *types.h* header file.

2.5 Interrupts

This section describes interrupt processing and interrupt configuration using the DSP56800E_Quick_Start tool. For detailed information on interrupts and interrupt processing for the 56F800E, please see the ***56800E 16-Bit Core Reference Manual*** and the target processor's User's Manual.

2.5.1 Processing Interrupts

An interrupt is an event that is generated by a condition inside the microcontroller or from external sources. When such event occurs, the interrupt processing transfers control from the currently executing program to an interrupt service routine (ISR), with the ability to later return to the current program upon completion of the ISR. Among the main uses of interrupts we can have data transfers between microcontroller memory and a peripheral device, or begin of execution of an algorithm upon reception of a new sample. An interrupt can also be used, for example, to exit the microcontroller's low-power wait processing state.

2.5.1.1 Interrupt Vector Table

The interrupt system on 56F800E can be defined as vectored. Each interrupt source has its own program memory location at a fixed address, to which program control is passed when an interrupt occurs. This program memory location must contain a JSR instruction with the address of the interrupt service routine (ISR). When this interrupt occurs, the JSR instruction is executed and the program control is passed to the ISR. The program memory containing the JSR instructions with the addresses of the ISR is called interrupt vector table.

Depending on processor configuration (the state of the EXTBOOT and EMI_MODE pins during reset), the interrupt vector table might be located at base address 0x0000 or 0x20000. During the code execution, the interrupt vector table base address can be changed by modifying the VBA register of interrupt controller unit (INTC).

In the DSP56800E_Quick_Start tool, the full interrupt vector table is always located at address 0x0000 regardless of the configuration of the EXTBOOT and EMI_MODE pins. The VBA register is set to zero during the startup code. For the case the processor configuration directs reset vector to 0x20000, the jump to startup code is also linked to this address. See Section 2.1.1 on page 2-2 for closer description of the booting process.

In the DSP56800E_Quick_Start tool, the interrupt vector table is implemented in C code which enables to effectively use the C preprocessor. The special macros defined in the global application configuration file (*appconfig.h*) can be used to setup the interrupt vector and to assign the interrupt priorities.

The interrupt controller and its configuration are described in more details later in Section 2.5.2.1.

2.5.1.2 Interrupt Processing Flow

Figure 2-2 shows an interrupt processing flow. The DSP56800E_Quick_Start tool does not provide any intermediate step when calling the ISR. When an interrupt occurs, the currently executed program is interrupted and the JSR instruction from the interrupt vector table is fetched. Executing the JSR instruction results in the program changing its flow directly to an ISR. Also the status register and the program counter are pushed onto the stack. When the user ISR finishes, it

executes a Return from Interrupt (RTI) instruction, which pops the program counter and the status register from the hardware stack. It puts the User Code back into the same state as it was in before execution, assuming that the User ISR saved and restored all the registers it had used.

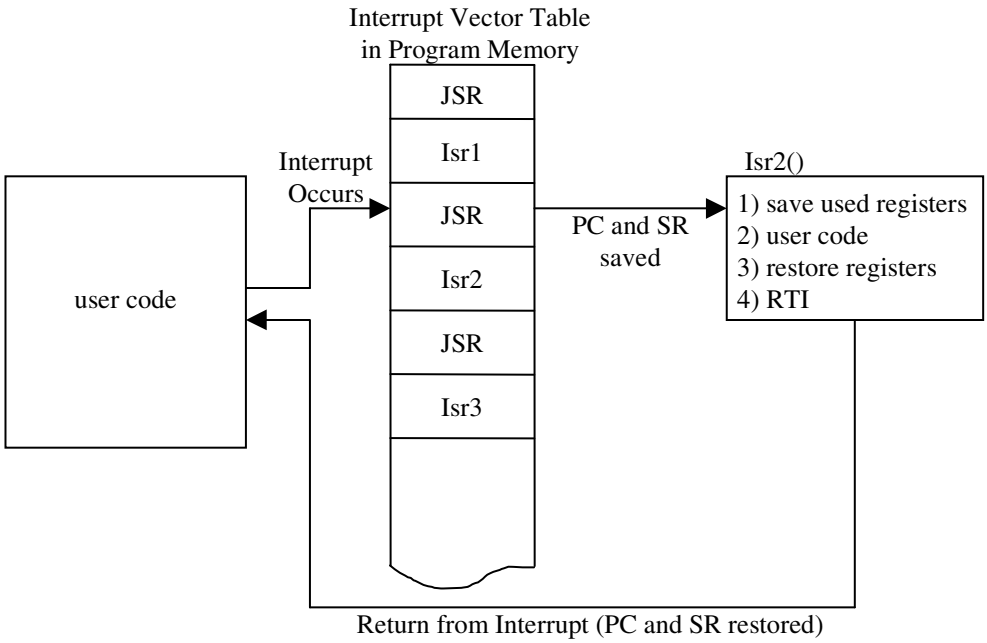


Figure 2-2. Interrupt Processing Flow

2.5.1.3 ISRs

An ISR is a program code that is executed when an interrupt is detected. An ISR is responsible for servicing the cause of the interrupt, such as reading a sample from a port when it is full or transmitting a sample to a port when it is empty. When an interrupt occurs, all other interrupts of the same or of a lower priority are disabled from executing, until the current ISR finishes executing. For this reason, an ISR should be as fast as possible to prevent any overflow or under run condition.

Inside the ISR it is necessary to save, and upon servicing the interrupt, to restore all used registers, including registers from the register bank used by the compiler. The DSP56800E_Quick_Start tool does not provide any automatic saving/restoring of used registers.

The last instruction of an ISR must be “Return from Interrupt” (RTI) instruction. This instruction restores the SR and the PC from the stack.

Both saving/restoring registers and using RTI instead of RTS are provided by the compiler directive *#pragma interrupt*. *#pragma interrupt* is used when declaring a C function and it instructs the compiler to save all registers used within a C function and to restore those register values upon exiting. Also it places an RTI instruction instead of an RTS at the end of the function. See Section 2.5.3.

2.5.1.4 Interrupt Priority Levels

On 56F800E hybrid microcontroller family, each interrupt can be assigned the interrupt priority level (IPL). It is the number from 0 (lowest priority) to 3 (highest priority). When servicing the interrupt, until the RTI instruction is executed, the other interrupts of the same and lower priority levels are masked (temporarily disabled). If there is an interrupt request of the masked priority level, its processing is postponed until the level is unmasked again.

This model assures that the interrupts of the same level can not “nest” one to each other. On the other hand, the higher priority interrupts do nest to the lower priority interrupts.

2.5.1.5 Fast Interrupts

Up to 2 interrupt sources can be declared as Fast Interrupts. The Fast Interrupts jump directly to a service routine based on values in the Fast Interrupt Vector Address registers without having to go to a jump table first.

IRQs used as fast interrupts MUST be set to priority level 2. Unexpected results can occur if a fast interrupt vector is set to any other priority.

Caution: A special Fast Interrupt Return instruction (*frtid*) must be used in order to return from the Fast Interrupt service routine. There are also several limitations in the way how the Fast Interrupt service routine can be coded. See the 56800E Processor Core Reference Manual for more details.

2.5.1.6 Clearing Interrupt Flags

Each on-chip peripheral interrupt source has its own interrupt flag, which must be cleared after the interrupt is serviced. For each peripheral module, the method of clearing the interrupt flag is different. As the DSP56800E_Quick_Start tool does not add any infrastructure code to the interrupt service routines, it also does not clear the interrupt flag inside the ISR. See Code Example 2-34.

Example 2-34. Clearing Interrupt Flags inside ISR

```
/*************************************************************************
      PWM A Reload Interrupt Service Routine
*************************************************************************/
#pragma interrupt
void pwmAReloadISR(void)
{
      /* ISR code */
      ...
      /* clear Reload interrupt flag */
      ioctl(PWM_A, PWM_CLEAR_RELOAD_FLAG, NULL);
}
```

This example shows the PWMA Reload Interrupt Service Routine. Note that the `PWM_CLEAR_RELOAD_FLAG` ioctl() command is used to clear the Reload Interrupt Flag (Bit 5) in the PWM Control Register and that this is the user’s responsibility.

2.5.2 Configuring Interrupts

This section describes the configuration of interrupts using the DSP56800E_Quick_Start tool. Interrupt configuration consists of installing the interrupt service routine (ISR) at the specified interrupt vector, enabling the interrupt and setting the interrupt priority level.

2.5.2.1 Installing ISRs

The DSP56800E_Quick_Start tool supports static (compile-time) installation of the ISRs, dynamic installation (run-time) is not supported. In general, static installation of ISRs requires less program memory and has a lower (or even no) time overhead.

The static installation of ISRs consists of writing the address of the ISR to the interrupt vector table for given interrupt source at compilation time. The interrupt vector table is located in the *vectors.c* file. By default all interrupt vectors are initialized with the address of the *unhandled_interrupt()* function in the *vectors.c* file, which contains the *debughlt* instruction and provides an alarm to the user, that this interrupt was not installed but has occurred, which is very useful when debugging. One exception to this is the Hardware Reset vector, which contains the address of the startup code - *Start()* routine in the *startup.c* file.

To install a user's ISR at the xxth interrupt vector add the following *#define* in *appconfig.h*:

```
#define  INT_VECTOR_ADDR_xx  userISRname
```

The conditional compilation then forces the compiler to use the *userISRname()* ISR instead of the default *unhandled_interrupt()* at the position of the xxth interrupt vector in the interrupt vector table. The *userISRname* is the placeholder for the name of interrupt service routine with prototype of

```
void userISRroutine (void)
```

In your source code, you then put the following code:

```
#pragma interrupt
void userISRname(void)
{
      /* ISR code */
}
```

The range of interrupt vectors which can be installed (xx) is 1 to 80. Vector 0 is the Hardware Reset vector and always refers to the *Start()* code.

2.5.2.2 Assigning Interrupt Priority Levels

As described in Section 2.5.1.4 on page 2-26, each enabled interrupt can be assigned to one interrupt priority level in range from 0 to 3.

There are some exceptions from this rule for the particular interrupt sources which has assigned a fixed priority levels. Also, as there are only two bits (four combinations) to encode five different states of the interrupt source (disabled, level 0,..., level 3) there is always one priority level, which cannot be set for any interrupt.

The DSP56800E_Quick_Start tool hides these difficulties and implementation details described above and simplifies the configuration of the interrupt priority levels to the maximal extent (while keeping the generated code optimal).

To enable the interrupt servicing and to assign a certain priority level, the user defines the macro:

```
INT_PRIORITY_LEVEL_xx INTC_LEVELn
```

To explicitly disable the interrupt, the user can define the macro as

```
INT_PRIORITY_LEVEL_xx INTC_DISABLED
```

where xx is the interrupt number (from 1 to 80) and n is the interrupt level (from 0 to 3). The interrupt sources configurations are then applied to a processor core by issuing the `INTC_INIT` ioctl command, for example in the *main* function.

The C preprocessor and compiler check the validity of the selected priority level early during the compilation and issues compilation errors if invalid combination of interrupt source number and interrupt priority level is requested (or if priority level is requested to be set for the source with fixed priority level).

2.5.2.3 Installing Fast Interrupts

As described in Section 2.5.1.5 on page 2-26, two interrupt sources can be selected as “Fast Interrupts”. For the fast interrupts, the interrupt controller does not fetch the *jsr* instruction from the vector table and directly loads the program counter (PC) with address specified in dedicated Fast Interrupt Vector Address (`FIVA`) registers.

In the DSP56800E_Quick_Start tool, the fast interrupts are automatically configured by the `INTC_INIT` code if the user defines the macros:

```
INTC_FIM0_INIT xx
```

or

```
INTC_FIM1_INIT xx
```

where xx specifies what interrupt source is to be selected as fast interrupt (0 or 1).

By default, the address of interrupt service routine defined by `INT_VECTOR_ADDR_xx` is then automatically loaded into the `FIVA` registers during the `INTC_INIT` command. The preprocessor also verifies the interrupt identified for a fast interrupt is configured to priority level 2 (which is required for the proper operation).

Caution: A special Fast Interrupt Return instruction (*frtid*) must be used in order to return from the Fast Interrupt service routine.

If there is another vector address to be used for the fast interrupt processing, instead of the default `INT_VECTOR_ADDR_xx`, the following macros can be defined in *appconfig.h*

```
INTC_FIVA0_INIT fastint0ISR
```

or

```
INTC_FIVA1_INIT fastint1ISR
```

where `fastint0ISR` and `fastint1ISR` are the placeholders for the fast interrupt service routine names.

2.5.2.4 Enabling Interrupts

In addition to the interrupt controller peripheral described above, the 56800E core has its own method how to enable and disable the interrupts of certain priority levels. So, regardless the interrupt setting defined by macros in *appconfig.h* and initialization done by the INTC_INIT command, there is another step to do to enable interrupt servicing in the application.

At the core level, the interrupts can be in four states:

- All interrupts disabled (default state) - use archDisableInt() macro
- All interrupts enabled - use archEnableInt() macro
- Priority levels 1, 2 and 3 enabled, level 0 disabled - use archEnableIntLvl123() macro
- Priority levels 2 and 3 enabled, levels 0 and 1 disabled - use archEnableIntLvl23() macro

2.5.3 Code Example

The following example shows the installation of the ISR into the interrupt vector table and shows how to enable interrupts using the DSP56800E_Quick_Start tool.

The following example shows the installation of the external interrupt $\overline{\text{IRQA}}$, the timer/counter D2 interrupt and the PWM A reload interrupt. The example shows a part of the code, which must be included in *appconfig.h*, all three ISRs and the initialization code. ISRs are declared as *#pragma interrupt* to instruct the compiler to save/restore all used registers and to terminate the ISRs with an RTI instruction. Inside the *appconfig.h* file, INT_VECTOR_ADDR_xx and ITCN_INT_PRIORITY_xx define statements are used to install the ISR at the specified interrupt vector and to define the interrupt priority level. The *achEnableInt()* macro and the ITCN driver commands ITCN_INIT_GPRS and ITCN_INIT_IPR are used to enable interrupts.

Example 2-35. Installing ISRs and enabling interrupts

1) *appconfig.h* file

```
/*************************************************************************
**
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*************************************************************************
**
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H
```

```
/*.***************************************************************************
*
*  File generated by Graphical Configuration Tool Mon, 26/Sep/2005, 11:28:11
*
*****************************************************************************.*
/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02010004L

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output Mode: Off: Tristated
.*/
#define SIM_GPS_INIT                     0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K:
Both bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K:
Lower byte enable
                     R/W: Read / Write , PS/DS select: DS only
```

```
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K:
Upper byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K:
Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                  0x0005
#define SEMI_CSBAR1_INIT                  0x0005
#define SEMI_CSBAR2_INIT                  0x0005
#define SEMI_CSOR0_INIT                   0x1FC3
#define SEMI_CSOR1_INIT                   0x1BA3
#define SEMI_CSOR2_INIT                   0x1DA3

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                    0x0000
#define INT_VECTOR_ADDR_17                irqA_isr
#define INT_PRIORITY_LEVEL_17             INTC_LEVEL1

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt:
Disable, Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                   0x1FC1


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

2) application code (*main.c* file)

```
/*****************************************************************************
**
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
**
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application demonstrating the use of external interrupt
*                      IRQA. Use IRQA button to toggle RED LED. GREEN LED is
flashing.
*
* TARGET: MC56F8346 device
*
*****************************************************************************
*/

#include "qs.h"

#include "occs.h"
#include "intc.h"
#include "gpio.h"

/* few EVM specific defines */

#define LED_RED BIT_0
#define LED_GREENBIT_2
#define LEDS (LED_RED | LED_GREEN)

/*****************************************************************************
**
       IRQA Interrupt service routine
*****************************************************************************
/

#pragma interrupt
void irqA_isr(void)
{
       /* toggle RED on port C */
       ioctl(GPIO_C, GPIO_TOGGLE_PIN, LED_RED);
}


/*****************************************************************************
**
       main
*****************************************************************************
*/
void main(void)
{
       int i;

       /* Setup LEDs GPIO on port C (could be done also via appconfig.h) */
       ioctl(GPIO_C, GPIO_SETAS_GPIO, LEDS);
       ioctl(GPIO_C, GPIO_SETAS_OUTPUT, LEDS);
       ioctl(GPIO_C, GPIO_WRITE_DATA, 0);

       /* configure Interrupt Controller (IPR) */
```

```
        ioctl(INTC, INTC_INIT, NULL);

        /* configure IRQ mode */
        ioctl(INTC, INTC_SELECT_EDGE_MODE, INTC_IRQA );

        /* enable maskable interrupts in Status Register (SR), bits I1 and I0 */
        archEnableInt();

        while (1)
        {
              /* keep GREEN flashing */
              ioctl(GPIO_C, GPIO_TOGGLE_PIN, LED_GREEN);

              for(i=0; i<100; i++)
                    archDelay(0xffff);
        }
}
```

2.6 Advanced Topics

This section describes the implementation details and the system code of each project created from the DSP56800E_Quick_Start tool stationery for the MC56F83xx hybrid controllers.

2.6.1 Project Targets

Each created project contains several targets for different hardware configurations of the microcontroller system. All targets are briefly described in the following tables:

Table 2-19. Targets of the MC56F8300DEMO project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	n/a	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	n/a	Stand Alone application
SDM_Simulator	Small	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core

Table 2-20. Targets of the MC56F8323EVM project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	n/a	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	n/a	Stand Alone application
SDM_Simulator	Small	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core

Table 2-21. Targets of the MC56F8346EVM project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
LDM_ExtRam	Large	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG3 off JG4 on	Standard Debug project with External Memory

Table 2-21. Targets of the MC56F8346EVM project.

LDM_xFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG3 on	Stand Alone application
LDM_pFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG3 on	Stand Alone application
LDM_IntRam	Large	Ext RAM (0x0000)	n/a	Int xRAM (0x0000)	n/a	Int RAM	JG3 off JG4 on	Debugging/Experimetnal (performance testing on separate P and X buses)
LDM_Simulator	Large	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core
SDM_ExtRam	Small	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG3 off JG4 on	Standard Debug project with External Memory
SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG3 on	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG3 on	Stand Alone application

Table 2-22. Targets of the MC56F8346CB project.

Target Name	**Data Model**	**Memory Used**					**EVM Board Jumpers**	**Target Description**
		Code	Boot Location	Data	Initial Data	Constant Data		
LDM_ExtRam	Large	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	J5 off J4 on	Standard Debug project with External Memory
LDM_xFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	J5 on	Stand Alone application
LDM_pFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	J5 on	Stand Alone application
LDM_IntRam	Large	Ext RAM (0x0000)	n/a	Int xRAM (0x0000)	n/a	Int RAM	J5 off J4 on	Debugging/Experimetnal (performance testing on separate P and X buses)
LDM_Simulator	Large	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core
SDM_ExtRam	Small	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	J5 off J4 on	Standard Debug project with External Memory
SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	J5 on	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	J5 on	Stand Alone application

Table 2-23. Targets of the MC56F8357EVM project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
LDM_ExtRam	Large	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG4 off JG5 on	Standard Debug project with External Memory
LDM_xFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG4 on	Stand Alone application
LDM_pFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG4 on	Stand Alone application
LDM_IntRam	Large	Ext RAM (0x0000)	n/a	Int xRAM (0x0000)	n/a	Int RAM	JG4 off JG5 on	Debugging/Experimetnal (performance testing on separate P and X buses)
LDM_Simulator	Large	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core
SDM_ExtRam	Small	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG4 off JG5 on	Standard Debug project with External Memory
SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG4 on	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG4 on	Stand Alone application

Table 2-24. Targets of the MC56F8367EVM project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
LDM_ExtRam	Large	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG4 off JG5 on	Standard Debug project with External Memory
LDM_xFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG4 on	Stand Alone application
LDM_pFlash	Large	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG4 on	Stand Alone application
LDM_IntRam	Large	Ext RAM (0x0000)	n/a	Int xRAM (0x0000)	n/a	Int RAM	JG4 off JG5 on	Debugging/Experimetnal (performance testing on separate P and X buses)
LDM_Simulator	Large	0x0000 (sim)	n/a	0x0000 (sim)	n/a	0x0000 (sim)	n/a	Software simulator of the processor core
SDM_ExtRam	Small	Ext RAM (0x0000)	0x0000	Ext RAM (0x2000)	n/a	Ext RAM (0x2000)	JG4 off JG5 on	Standard Debug project with External Memory

Table 2-24. Targets of the MC56F8367EVM project.

SDM_xFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	xFlash	xFlash	JG4 on	Stand Alone application
SDM_pFlash	Small	pFlash (0x0000)	0x20000	Int xRAM (0x0000)	pFlash	xFlash	JG4 on	Stand Alone application

Table 2-25. Targets of the MC56F8013DEMO and MC56F8014DEMO project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_pFlash	Small	pFlash (0x0000)	0x0000	Int xRAM (0x0000)	pFlash	Int xRAM (before Data)	n/a	Stand Alone application
SDM_Simulator	Small	0x0000 (sim)	n/a	0x0000 (sim)	n/a	before data	n/a	Software simulator of the processor core

Table 2-26. Targets of the MC56F8013CB project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_pFlash	Small	pFlash (0x0000)	0x0000	Int xRAM (0x0000)	pFlash	Int xRAM (before data)	n/a	Stand Alone application
SDM_Simulator	Small	0x0000 (sim)	n/a	0x0000 (sim)	n/a	before data	n/a	Software simulator of the processor core

Table 2-27. Targets of the MC56F8023DEMO, MC56F8023CB and MC56F8025DEMO project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_pFlash	Small	pFlash (0x4000)	0x4000	Int xRAM (0x0000)	pFlash	Int xRAM (before data)	n/a	Stand Alone application
SDM_Simulator	Small	0x4000 (sim)	n/a	0x0000 (sim)	n/a	before data	n/a	Software simulator of the processor core

Table 2-28. Targets of the MC56F8037EVM project.

Target Name	Data Model	Memory Used					EVM Board Jumpers	Target Description
		Code	Boot Location	Data	Initial Data	Constant Data		
SDM_pFlash	Small	pFlash (0x0000)	0x0000	Int xRAM (0x0000)	pFlash	Int xRAM (before data)	n/a	Stand Alone application
SDM_Simulator	Small	0x0000 (sim)	n/a	0x0000 (sim)	n/a	before data	n/a	Software simulator of the processor core

There is a different linker command file (LCF) for each target, which defines the destination memory ranges used by the linker. Although the syntax of the LCF and C header files are completely different. The LCF for each target is also used as prefix[1] header file in its target configuration. The macros defined in the LCF identify the target for further conditional compilation of the project source files.

The trick which enables using a file with the LCF syntax as a header file is shown on Code Example 2-36. It successfully exploits the fact that the '#' sign is treated as a start of comment line in LCF syntax, so the C-like #define statements do not cause the LCF syntax errors. On the other side, the #if 0 ... #endif block excludes the LCF part of the file from C compilation.

1. Prefix file is unconditionaly included at the begining of every compiled C file.

Example 2-36. LDM_ExtRam.cmd linker command file

```
#include "version.h"
#include "hawkcpu.h"

#define TARGET_LDM               /* Large Data Model */
#define TARGET_CODE_EXTRAM       /* Code located in external pRAM */
#define TARGET_CONSTDATA_EXTRAM  /* Constants and const s located in external xRAM */
#define TARGET_INITDATA_EXTRAM   /* Initialized global vars located in external RAM */
#define TARGET_DATA_EXTRAM       /* Variables located in external RAM */

#pragma define_section fardata "fardata.data" "fardata.bss" RW
#pragma define_section pramcode "pramcode.text" RWX

#if 0  /* everything below is excluded from C compilation */

MEMORY
{
...
}

SECTIONS
{
...
}

...

#endif /* end of code excluded by C-preprocessor */
```

2.6.2 Inside Startup Code

This section goes step-by-step through the processor initialization code described briefly in Section 2.1.2 on page 2-3. The startup code described here can be found in the *startup.c* file, located in the *SystemConfig* subdirectory of any project created using the DSP56800E_Quick_Start tool stationery.

2.6.2.1 Symbols Used in Startup Code

2.6.2.1.1 Included Header Files

The master Quick_Start header file *qs.h* is included in the startup code. This file further includes other critical system files to define common C types, peripheral module base addresses and other types and macros required by the startup code. The application configuration header file *appconfig.h* is also included so the startup code is able to configure system modules like OCCS (PLL) and SEMI.

```
#include "qs.h"
```

2.6.2.1.2 Initial Value of Operation Mode Register (OMR)

Although it is not very common, the initial value of the Operation Mode Register (OMR) can be specified in *appconfig.h* using the `OMR_INIT` macro.

The following *startup.c* statements define the default initial OMR value for the cases when the user had not defined the `OMR_INIT` in *appconfig.h*:

```
#ifndef TARGET_OMR_INIT
```

```
#define TARGET_OMR_INIT 0
#endif

#ifndef OMR_INIT
#define OMR_INIT 0|(TARGET_OMR_INIT)
#endif
```

The default initialization value of the OMR is based on the TARGET_OMR_INIT value, which might be defined in the prefix file (LCF) within the active compilation target. Currently, the TARGET_OMR_INIT is not defined in the prefix file of any target, leaving the initial OMR value on 0x0000.

The following OMR bits are important for the proper operation of the C application:

- CM = 0 - optional for C application
- XP = 0 - enabling separate program and data buses (Harvard Architecture)
- R = 0 - rounding off, required for C applications
- SA = 0 - saturation off, required for C applications
- EX = 0 - complete X memory space as external, required by CodeWarrior debugger

The critical OMR bits are checked by the C preprocessor directive, issuing the compile-time warning when found in the OMR_INIT value:

```
#if (OMR_INIT & (OMR_CM|OMR_XP|OMR_R|OMR_SA))
#warning Initial OMR value might be invalid for the C project
#endif
#if (OMR_INIT) & OMR_EX
#warning  CodeWarrior cannot debug projects with OMR.EX bit set
#endif
```

2.6.2.1.3 Other appconfig.h Symbols

Using the OCCS_REQUIRED_LOCK_MODE macro, the user specifies in which lock state of the PLL the setup code continues to the rest of the startup code:

- 0x20 (default) - continue when “coarse” lock mode is reached (bit LCK0 in PLLSR)
- 0x40 - continue when “fine” lock mode is reached (bit LCK1 in PLLSR)

```
#ifndef OCCS_REQUIRED_LOCK_MODE
#define OCCS_REQUIRED_LOCK_MODE 0x20 /* coarse (LCK0) by default */
#endif
#if (OCCS_REQUIRED_LOCK_MODE != 0x40) && (OCCS_REQUIRED_LOCK_MODE != 0x20)
#error OCCS_REQUIRED_LOCK_MODE must be one of 0x20 (coarse) or 0x40 (fine)
#endif
```

One of the startup code optional features is to perform the internal data RAM checking. The checking algorithm, fully described later in Section 2.6.2.2.7, uses two values which writes, reads and verifies to check each memory location. By default the two values are 0xAAAA and 0x5555. If there is any reason to change the values, the user can define the macros CONFIG_INTRAM_CHECKVALUE1 and CONFIG_INTRAM_CHECKVALUE2 in the *appconfig.h* file.

```
#ifndef CONFIG_INTRAM_CHECKVALUE1
#define CONFIG_INTRAM_CHECKVALUE1 0xaaaa
#endif
#ifndef CONFIG_INTRAM_CHECKVALUE2
#define CONFIG_INTRAM_CHECKVALUE2 0x5555
```

```
#endif
```

2.6.2.1.4 Linker Command File Symbols

While linking, the linker replaces any zeros generated by compiler for *external* symbols with proper values calculated during linking process when the physical addresses of the symbols are known. Some values of external symbols can be also specified directly by the directives in the linker command file.

The following symbols are specified by the LCF and provide physical address of memory segments used in the startup code:

```
/* external constants defined in LCF */
extern _Lstack_addr;
extern _Ldata_size;
extern _Ldata_ROM_addr;
extern _Ldata_RAM_addr;
extern _Ldata2_size;
extern _Ldata2_ROM_addr;
extern _Ldata2_RAM_addr;
extern _Ldatap_size;
extern _Ldatap_ROM_addr;
extern _Ldatap_RAM_addr;
extern _Lbss_size;
extern _Lbss_start;
extern _Lbss2_size;
extern _Lbss2_start;
extern _Lbssp_size;
extern _Lbssp_start;
extern _Linternal_RAM_addr;
extern _Linternal_RAM_size;
extern _Linterrupt_vectors_addr;
```

2.6.2.2 Startup Source Code

The following subsections describe the source code of the Start assembly function.

```
asm void Start(void)
{
```

2.6.2.2.1 Initialize Interrupt Vectors Base Address

Depending on the state of the EXTBOOT and EMI_MODE pins during the system reset, the vector table is located at the beginning of the Boot Program Flash or (if Boot Flash is not available) at the beginning of the Program RAM. The startup code always updates the Vector Table Base Address (VBA) to beginning of ".interrupt_vectors" section where the Quick_Start vector table is located. By default this table is always put to the beginning of the Program RAM anyway.

By defining the ARCH_VECTBL_ADDR macro in the *appconfig.h* configuration file, the VBA may be forced to a custom value.

```
/* relocate vector table properly */
    #ifdef ARCH_VECTBL_ADDR
    move.l ARCH_VECTBL_ADDR,A
    #else
    move.l #_Linterrupt_vectors_addr,A
    #endif
    asrr.l #7,A
    move.w A0,ArchIO.Intc.vba
```

2.6.2.2.2 Clear COP Counter and Keep Clearing Values in Registers

On the newer 56F800E-based devices, the COP Watchdog counter is enabled after reset, so it is necessary to clear this counter periodically during any lengthy operation in the startup code. Early in the startup, the COP counter is initially cleared and the clearing values are preserved in registers. The R5, C1 and D1 registers are not changed anywhere in the rest of the startup code and are used to clear the COP without loading the constant values again.

```
/* clear COP watchdog counter, keep clearing values in registers C1,D1,R5 */
    moveu.w #ArchIO.Cop.copctr,R5
    move.w 0x5555,C1
    move.w 0xAAAA,D1
    move.w C1,X:(R5)
    move.w D1,X:(R5)
```

2.6.2.2.3 Setup the Operation Mode Register (OMR)

The “one” bits in the OMR_INIT value are set in the Operating Mode Register.

```
/* setup the OMR */
       bfset OMR_INIT,omr
       nop
       nop
```

2.6.2.2.4 Other Initialization

The M01 register is initialized to -1 to activate linear addressing mode with R0 and R1 registers.

```
/* setup the m01 register for linear addressing */
       move.w #-1,x0
       moveu.w x0,m01
```

The values on the Hardware Stack are cleared (for proper debugger behavior).

```
/* clear (read-out) the hardware stack */
       moveu.w hws,la
       moveu.w hws,la
       nop
       nop
```

2.6.2.2.5 Core Clock Setup (OCCS)

The PLL Oscillator Control Register and the Divide-By Register are initialized with the *appconfig.h* values if defined. The value in the Divide-By Register controls the prescaler and postscaler frequency divisors and also multiplication factor of the PLL. Note that before the PLLCR register is written, the PLL remains turned off and the system clock is still taken from a default clock source (now divided by prescaler value). The default clock source is an external oscillator or an internal relaxation oscillator on some devices.

```
/* configure external oscillator and clock mode */

#ifdef OCCS_OSCTL_INIT
#define OSCTL_TEMP (OCCS_OSCTL_INIT & 0x3fff) /* keep internal osc. enabled */
    move.w  #OSCTL_TEMP,ArchIO.Pll.osctl      /* OSCTL,even if PLL not used */
    nop
    nop
#endif

/* setup the PLL according to appconfig.h values */

#ifdef OCCS_PLLDB_INIT
    move.w  OCCS_PLLDB_INIT,ArchIO.Pll.plldb  /* PLLDB, even if PLL not used */
    nop
    nop
#endif
```

On the devices equipped with an internal relaxation oscillator, a user may want to initialize the trimming value in the Oscillator Control Register by the factory-measured value which is saved in the read-only Flash area (FMOPT1 Register).

```
/* load factory trimming value of the internal relaxation oscillator? */

#if OCCS_USE_FACTORY_TRIM
    move.w  ArchIO.Pll.osctl,x0
    move.w  ArchIO.Hfm.fmopt1,y0
    bfclr   #0x03ff,x0
    bfclr   #0xfc00,y0
    or.w    y0,x0
    move.w  x0,ArchIO.Pll.osctl
    nop
#endif
```

Then, if the PLL Control Register initial value is defined in *appconfig.h*, the PLL setup code is executed:

```
#ifdef OCCS_PLLCR_INIT
```

On the new devices (e.g. 56F802x/3x), all pins are in the GPIO mode after reset, including the pins which may be needed as an external clock or crystal source. The startup code automatically re-configures these pins to the required clock-related mode before switching to an external clock. The new devices are identified by OCCS version 3 and SIM version 4 in the new code.

NOTE:

The peripheral module version identifiers are defined in the *arch.h* file for each device purely for an internal use in the DSP56800E_Quick_Start code. The version numbers **do not** rely to chip or silicon version.

```
/* on new devices, some external pins may be needed if PLLCR.PRESC=1 */

#if defined(OCCS_VERSION_3)&& defined(SIM_VERSION_4) && (OCCS_PLLCR_INIT & 0x4)

    /* first get EXT_SEL and CLK_MODE values (see OSCTL register) */
    #ifdef OCCS_OSCTL_INIT
    #define _OCCS_EXTSEL  (((OCCS_OSCTL_INIT) >> 10) & 0x3)
    #define _OCCS_CLKMODE (((OCCS_OSCTL_INIT) >> 12) & 0x1)
    #else
    #define _OCCS_EXTSEL  0 /* reset value is 0 */
    #define _OCCS_CLKMODE 1 /* reset value is 1 */
    #endif

    /* primary external clock on GPIO_B6 */
    #if _OCCS_EXTSEL == 0
    bfset 0x4000, ArchIO.Sim.sim_gpsb0
    bfclr 0x2000, ArchIO.Sim.sim_gpsb0
    bfset 0x0040, ArchIO.PortB.per

    /* alternate external clock on GPIO_B5 */
    #elif _OCCS_EXTSEL == 1
    bfset 0x1000, ArchIO.Sim.sim_gpsb0
    bfclr 0x0800, ArchIO.Sim.sim_gpsb0
    bfset 0x0020, ArchIO.PortB.per

    /* external clock on XTAL (GPIO_D5)*/
    #elif _OCCS_CLKMODE
    bfset 0x1000, ArchIO.Sim.sim_gpscd
```

```
    bfset 0x0020, ArchIO.PortD.per

    /* crystal on EXTAL and XTAL pins (GPIO_D4 and GPIO_D5)*/
    #else
    bfclr 0x1000, ArchIO.Sim.sim_gpscd
    bfset 0x0030, ArchIO.PortD.per

    /* give it some time until crystal/resonator stabilizes */
    /* we now run from internal relaxation oscillator / 2 (i.e. 4MHz) */
    move.w #5000,x0   /* wait 50ms */
    do x0,waitosc
    rep 36; nop;      /* sigle loop pass takes 40 cycles */
    move.w C1,X:(R5)  /* and also clears the watchdog */
    move.w D1,X:(R5)
waitosc:

    #endif /* _OCCS_EXTSEL cases */

    /* switch to external clock source (set PRESC=1) */
    nop;
    bfset 0x4,ArchIO.Pll.pllcr
    nop
    nop

#endif /* defined(OCCS_VERSION_3) && (OCCS_PLLCR_INIT & 0x4) */
```

When the PLL is to be turned on, it is first decided whether the code is running on real chip or in software simulator. The simulator mode is identified by looking at the “clock source” bit-field value in the PLLCR register. In the simulator mode, the PLL setup is skipped because the loop waiting for the PLL lock would never finish.

```
#if ((OCCS_PLLCR_INIT & 3) == 2) /* PLL active ? */

#define PLLCR_TEMP (OCCS_PLLCR_INIT & 0xfc | 0x01)
                                 /* interrupts off and PLL bypassed */

    brclr 1,ArchIO.Pll.pllcr,skip_pll_lock /* skip PLL in simulator mode */
```

While still running from an external oscillator, the PLL lock detector is activated and it is waiting until the PLL lock is detected. According to *appcofing.h* setting, the required lock state is either “coarse” or “fine” - which corresponds to the PLLSR bits LCK1 and LCK0. Note that the COP counter is periodically cleared while waiting in the loop.

```
      move.w  #PLLCR_TEMP,ArchIO.Pll.pllcr   /* PLL lock detector ON, core
                                             still on prescaler */
pll_lock:
   move.w C1,X:(R5)  /* clear COP watchdog counter while waiting in the loop */
    move.w D1,X:(R5)

    brclr   OCCS_REQUIRED_LOCK_MODE,ArchIO.Pll.pllsr,pll_lock   /* test lock */
```

When the PLL is locked, the system clock is switched to PLL and the PLLCR is finally initialized with the user defined value. As the last, the pending PLL interrupts are cleared.

```
    nop
    nop
    move.w  OCCS_PLLCR_INIT,ArchIO.Pll.pllcr  /* PLL locked: final PLL setup */

skip_pll_lock:
    move.w  ArchIO.Pll.pllsr,x0     /* clear pending clkgen interrupts */
    move.w  x0,ArchIO.Pll.pllsr
```

```
    nop
```

If the PLL is not to be enabled, the initial value of the PLL Control Register is simply written.

```
#else /* ((OCCS_PLLCR_INIT & 3) == 2) PLL not active */

      move.w  OCCS_PLLCR_INIT,ArchIO.Pll.pllcr /* write PLLCR init value */

#endif /* ((OCCS_PLLCR_INIT & 3) == 2) */
#endif /* OCCS_PLLCR_INIT */
```

2.6.2.2.6 External Memory Interface Setup (SEMI)

Before the data memory is initialized (the bss segment is cleared and the global variables segment is copied from non-volatile memory), the external memory interface (SEMI) has to be first set up. The initialization is done by copying the initial values defined in *appconfig.h* to the SEMI peripheral registers.

```
#ifdef SEMI_BASE

    moveu.w #ArchIO.Semi.csbar, R0
    moveu.w #ArchIO.Semi.csor,  R1
    moveu.w #ArchIO.Semi.cstc,  R2

#ifdef SEMI_CSBAR0_INIT
    move.w SEMI_CSBAR0_INIT, X:(R0+0)
#endif
#ifdef SEMI_CSOR0_INIT
    move.w SEMI_CSOR0_INIT, X:(R1+0)
#endif
#ifdef SEMI_CSTC0_INIT
    move.w SEMI_CSTC0_INIT, X:(R2+0)
#endif
#ifdef SEMI_CSBAR1_INIT
    move.w SEMI_CSBAR1_INIT, X:(R0+1)
#endif
#ifdef SEMI_CSOR1_INIT
    move.w SEMI_CSOR1_INIT, X:(R1+1)
#endif
#ifdef SEMI_CSTC1_INIT
    move.w SEMI_CSTC1_INIT, X:(R2+1)
#endif
```

The memory chip selects signals CS2 to CS7 are shared with GPIO port D pins. So, when any external memory is to be configured on these banks using the values from the *appconfig.h* file, the appropriate GPIO circuitry is turned off here by setting the Peripheral Enable Register of the GPIO port D.

```
#ifdef SEMI_CSBAR2_INIT
    move.w SEMI_CSBAR2_INIT, X:(R0+2)
#endif
#ifdef SEMI_CSOR2_INIT
    move.w SEMI_CSOR2_INIT, X:(R1+2)
    bfset 0x0001, ArchIO.PortD.per      /* enable CS2 on GPIO PD0 */
#endif
#ifdef SEMI_CSTC2_INIT
    move.w SEMI_CSTC2_INIT, X:(R2+2)
#endif
#ifdef SEMI_CSBAR3_INIT
    move.w SEMI_CSBAR3_INIT, X:(R0+3)
#endif
#ifdef SEMI_CSOR3_INIT
    move.w SEMI_CSOR3_INIT, X:(R1+3)
    bfset 0x0002, ArchIO.PortD.per      /* enable CS3 on GPIO PD1 */
```

```
#endif
#ifdef SEMI_CSTC3_INIT
    move.w SEMI_CSTC3_INIT, X:(R2+3)
#endif
#ifdef SEMI_CSBAR4_INIT
    move.w SEMI_CSBAR4_INIT, X:(R0+4)
#endif
#ifdef SEMI_CSOR4_INIT
    move.w SEMI_CSOR4_INIT, X:(R1+4)
    bfset 0x0004, ArchIO.PortD.per      /* enable CS4 on GPIO PD2 */
#endif
#ifdef SEMI_CSTC4_INIT
    move.w SEMI_CSTC4_INIT, X:(R2+4)
#endif
#ifdef SEMI_CSBAR5_INIT
    move.w SEMI_CSBAR5_INIT, X:(R0+5)
#endif
#ifdef SEMI_CSOR5_INIT
    move.w SEMI_CSOR5_INIT, X:(R1+5)
    bfset 0x0008, ArchIO.PortD.per      /* enable CS5 on GPIO PD3 */
#endif
#ifdef SEMI_CSTC5_INIT
    move.w SEMI_CSTC5_INIT, X:(R2+5)
#endif
#ifdef SEMI_CSBAR6_INIT
    move.w SEMI_CSBAR6_INIT, X:(R0+6)
#endif
#ifdef SEMI_CSOR6_INIT
    move.w SEMI_CSOR6_INIT, X:(R1+6)
    bfset 0x0010, ArchIO.PortD.per      /* enable CS6 on GPIO PD4 */
#endif
#ifdef SEMI_CSTC6_INIT
    move.w SEMI_CSTC6_INIT, X:(R2+6)
#endif
#ifdef SEMI_CSBAR7_INIT
    move.w SEMI_CSBAR7_INIT, X:(R0+7)
#endif
#ifdef SEMI_CSOR7_INIT
    move.w SEMI_CSOR7_INIT, X:(R1+7)
    bfset 0x0020, ArchIO.PortD.per      /* enable CS7 on GPIO PD5 */
#endif
#ifdef SEMI_CSTC7_INIT
    move.w SEMI_CSTC7_INIT, X:(R2+7)
#endif
```

As the last, the SEMI Bus Control Register is initialized.

```
/* global wait states not covered by CS registers */
#ifdef SEMI_BCR_INIT
    move.w SEMI_BCR_INIT, ArchIO.Semi.bcr
#endif

#endif /* SEMI_BASE */
```

2.6.2.2.7 Internal Memory Checking

Checking the internal data RAM is an optional feature of the startup code. When the `INTXRAM_CHECK_ENABLED` macro is defined in *appconfig.h*, this feature is activated.

The memory checking process consists of tree parts:

- Complete memory fill (value 0xAAAA) & read + compare
- Single write, read & compare for each memory location

- Two immediate reads from different memory locations & compare

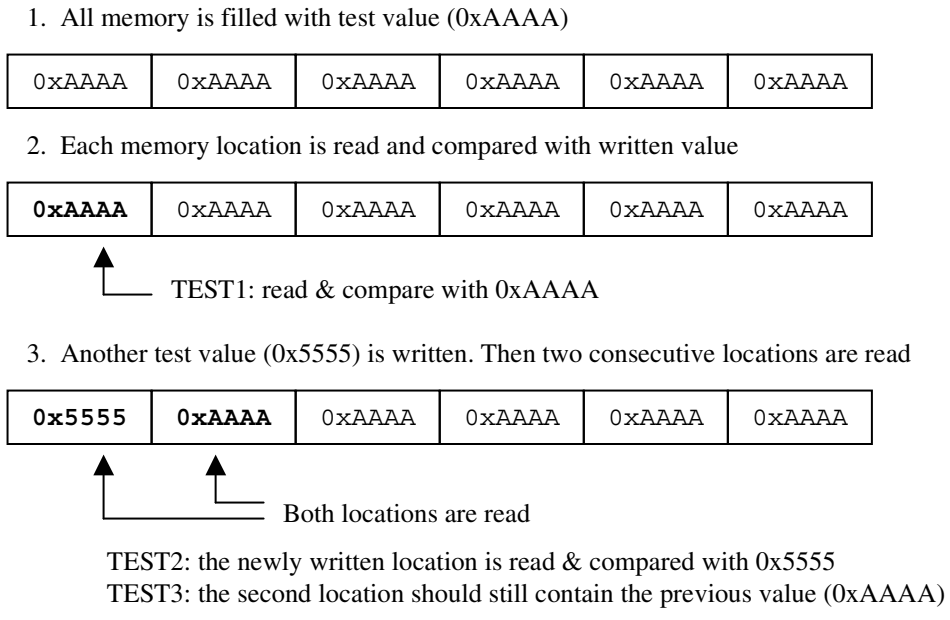


Figure 2-3. Memory Checking Process

In case if any of the three tests fails, the application execution is halted by *debughlt* and *stop* instructions.

The compile-time warning is issued when the internal memory checking is activated in targets that do not use the internal memory.

```
      /* internal RAM memory test */

      #ifdef INTXRAM_CHECK_ENABLED
      #ifndef TARGET_DATA_INTRAM
      #warning Internal Memory Checking is active but variables go elsewhere
      #endif

      move.l  #>>_Linternal_RAM_addr,r1      /* memory pointer */
      move.l  #>>_Linternal_RAM_size,r2      /* memory size */
      move.w  CONFIG_INTRAM_CHECKVALUE1, x0  /* x0=write/test value 1 */
      move.w  CONFIG_INTRAM_CHECKVALUE2, y0  /* y0=write/test value 2; */
      move.w  #0,b                           /* b0=0, b1 will be used as "b" */

      rep r2; move.w x0,x:(r1)+              /* fill memory with value test1 */

      move.l  #>>_Linternal_RAM_addr,r1      /* initialize verify memory pointer */
      do      r2,end_intramcheck1            /* start verify loop */


          move.w C1,X:(R5)                       /* clear COP watchdog counter */
          move.w D1,X:(R5)

      cmp.w   x:(r1),x0                      /* TEST1: read & compare */
      beq <t1passed                          /* TEST1: OK ? */
      debughlt

t1passed:
      move.w  y0,x:(r1)          /* TEST2: write test2 */
      move.w x:(r1+1),y1         /* read from incremented address (see TEST3) */
```

```
        cmp.w   x:(r1),y0           /* read written value & compare (should be test2) */
        beq <t2passed               /* TEST2: OK ? */

        nop
        debughlt /* !! MEMORY TEST FAILED !! */
        stop

t2passed:
        move.w  lc,b        /* skip TEST3 for the last memory cell (when LC==1) */
        cmp.w   #1,b
        ble    <t3passed
        cmp.w  y1,x0        /* TEST3: value from incremented addr. should be ==test1 */
        beq <t3passed       /* TEST3: OK ? */

        nop
        debughlt /* !! MEMORY TEST FAILED !! */
        stop

t3passed:
        move.w  b0,x:(r1)+          /* clear checked memory location */
        nop
        nop                         /* without nops, the branch to t3passed above */
        nop                         /* could confuse the hardware loop unit */

end_intramcheck1:
        #endif
```

2.6.2.2.8 Stack Pointer Initialization

The stack pointer (SP) register is initialized to the first odd value after _Lstack_addr symbol generated by linker command file. The first stack location is then initialized to NULL.

```
        /* initialize stack */

              move.l #>>_Lstack_addr,r0
              bftsth #$0001,r0
              bcc <noinc
              adda #1,r0
        noinc:
              tfra   r0,sp
              move.w #0,r1
              nop
              move.w r1,x:(sp)
              adda   #1,sp
```

2.6.2.2.9 Clearing .bss, .bss.pmem and fardata.bss Segments

The *.bss* is the memory segment containing the global or static C variables to which are not assigned initial values (or the initial value is 0). This segment is cleared by the startup code so the global and static C variables are initialized to 0.

Note that the COP counter is periodically cleared in all loops below.

```
        /* clear BSS segment (can't use 'do' and its 16 bit loop counter) */

              move.l  #>>_Lbss_size,r2  /* bss size */
              tsta.l  r2
              beq <end_clearbss;        /* skip if size is 0 */
              move.l  #>>_Lbss_start,r1 /* dest address */
              move.w  #0,x0
        loop_clearbss:
              move.w  C1,X:(R5)               /* clear COP watchdog counter */
              move.w  D1,X:(R5)
              move.w  x0,x:(r1)+        /* clear value at r1 */
```

```
        dectsta r2;                /* long loop counter */
        bne <loop_clearbss;
end_clearbss:
```

The same is done with the *.bss* segment of the fardata section (addresses after 0x10000). The full name of segment is *.fardata.bss*. In the startup code it is referenced as *bss2*.

```
/* clear BSS2 segment (can't use 'do' and its 16 bit loop counter) */

        move.l  #>>_Lbss2_size,r2 /* bss size */
        tsta.l  r2
        beq <end_clearbss2;       /* skip if size is 0 */
        move.l  #>>_Lbss2_start,r1 /* dest address */
        move.w  #0,x0
loop_clearbss2:
        move.w  C1,X:(R5)               /* clear COP watchdog counter */
        move.w  D1,X:(R5)
        move.w  x0,x:(r1)+        /* clear value at r1 */
        dectsta r2;               /* long loop counter */
        bne <loop_clearbss2;
end_clearbss2:
```

And again the process is repeated for the Program RAM-based variables. In the startup code, this *.bss.pmem* segment is referenced as *bssp*. Note that the move instruction accesses the P (program) space.

```
/* clear BSSP (program RAM) segment (can't use 'do' and its 16 bit loop
counter) */

        move.l  #>>_Lbssp_size,r2        /* bssp size */
        tsta.l  r2
        beq <end_clearbssp;              /* skip if size is 0 */
        move.l  #>>_Lbssp_start,r1       /* dest address */
        move.w  #0,x0
loop_clearbssp:
        move.w  C1,X:(R5)               /* clear COP watchdog counter */
        move.w  D1,X:(R5)
        move.w  x0,p:(r1)+               /* clear value at r1 */
        dectsta r2;                      /* long loop counter */
        bne <loop_clearbssp;
end_clearbssp:
```

2.6.2.2.10 Initializing Global Variables

The C variables to which are assigned a non-zero initial values must be initialized using the data from a non-volatile memory. Using the directives in the linker command file, the initial data are stored in the internal Flash memory. The source and destination addresses are calculated by the linker and exported as symbols.

Depending on the application target selected (see Section 2.6.1 on page 2-34) the internal X or P Flash memory are used and the appropriate code is compiled.

Note that the COP counter is periodically cleared in all loops below.

```
/* copy variable initialization data from xFlash to destination (16bit LC ok)*/

#ifdef TARGET_INITDATA_XFLASH
    move.l  #>>_Ldata_size,r2/* set data size */
    move.l  #>>_Ldata_ROM_addr,r3/* src address -- xROM data start */
    move.l  #>>_Ldata_RAM_addr,r1/* dest address -- xRAM data start */
    do      r2,end_xrom2xram/* copy for r2 times */
    move.w  C1,X:(R5)               /* clear COP watchdog counter */
    move.w  D1,X:(R5)
move.w  x:(r3)+,x0        /* fetch value at address r3 */
```

```
    move.w  x0,x:(r1)+    /* stash value at address r1 */
end_xrom2xram:

/* dtto for fardata (if available) */

#ifdef SEMI_BASE
    move.l  #>>_Ldata2_size,r2/* set data size */
    move.l  #>>_Ldata2_ROM_addr,r3/* src address -- xROM data start */
    move.l  #>>_Ldata2_RAM_addr,r1/* dest address -- xRAM data start */
    do      r2,end_xrom2xram2/* copy for r2 times */
    move.w  C1,X:(R5)               /* clear COP watchdog counter */
    move.w  D1,X:(R5)
move.w  x:(r3)+,x0        /* fetch value at address r3 */
    move.w  x0,x:(r1)+    /* stash value at address r1 */
end_xrom2xram2:
#endif
#endif

/* copy variable initialization data from pFlash to destination (long loop) */

#ifdef TARGET_INITDATA_PFLASH
    move.l  #>>_Ldata_size,r2/* set data size */
    tsta.l  r2
    beq <end_prom2xram
    move.l  #>>_Ldata_ROM_addr,r3/* src address -- xROM data start */
    move.l  #>>_Ldata_RAM_addr,r1/* dest address -- xRAM data start */
loop_prom2xram:
    move.w  C1,X:(R5)               /* clear COP watchdog counter */
    move.w  D1,X:(R5)
    move.w  p:(r3)+,x0    /* fetch value at address r3 */
    move.w  x0,x:(r1)+    /* stash value at address r1 */
    dectsta r2
    bne <loop_prom2xram
end_prom2xram:

/* dtto for fardata (if available) */

#ifdef SEMI_BASE
    move.l  #>>_Ldata2_size,r2/* set data size */
    tsta.l  r2
    beq <end_prom2xram2
    move.l  #>>_Ldata2_ROM_addr,r3/* src address -- xROM data start */
    move.l  #>>_Ldata2_RAM_addr,r1/* dest address -- xRAM data start */
loop_prom2xram2:
    move.w  C1,X:(R5)               /* clear COP watchdog counter */
    move.w  D1,X:(R5)
    move.w  p:(r3)+,x0    /* fetch value at address r3 */
    move.w  x0,x:(r1)+    /* stash value at address r1 */
    dectsta r2
    bne <loop_prom2xram2
end_prom2xram2:
#endif
#endif
```

Next, the initialized program-RAM variables (those not in *.bss.pmem* section) and also the program-RAM-based code (from the Quick_Start-specific *pramcode* section) is initialized using the values from the program Flash.

```
/* in any flash-based target, do copy pram-variable initialization data
   (and ram-based code) from pFlash storage to destination in program-ram */

#ifdef TARGET_CODE_PFLASH
    move.l  #>>_Ldatap_size,r2/* set data size */
    tsta.l  r2
    beq <end_prom2pram
    move.l  #>>_Ldatap_ROM_addr,r3/* src address -- pROM data start */
```

```
    move.l  #>>_Ldatap_RAM_addr,r1/* dest address -- pRAM data start */
loop_prom2pram:
    move.w  C1,X:(R5)               /* clear COP watchdog counter */
    move.w  D1,X:(R5)
    move.w  p:(r3)+,x0    /* fetch value at address r3 */
    move.w  x0,p:(r1)+    /* stash value at address r1 */
    dectsta r2
    bne <loop_prom2pram
end_prom2pram:
#endif
```

To find out more details about the *fardata* and *pramcode* sections briefly mentioned above, please see the *fardata_demo* sampe application, available for example for 56F8346 and higher devices.

2.6.2.2.11 Calling the main()

As the last step of the startup code, the *userPreMain* file and the *main* functions are called. The *userPreMain* can be found in the *arch.c* file and contains architecture and peripheral specific initialization code. It is empty in the current implementation.

In the case the *main* is prototyped with standard *argc* and *argv* arguments, the 0 and NULL values are passed - as it makes no sense to use them in the embedded application.

```
/* call userPreMain() from appconfig.c */
      jsr userPreMain

/* call main() */
      move.w #0,y0                  /* pass parameters to main() */
      move.w #0,R2
      move.w #0,R3

      jsr main                      /* call the user program */
```

2.6.2.2.12 Never-Reached Finish Code

In case that the main ever returns - which would be very uncommon case - the *userPostMain* de-initialization code (empty) from *arch.c* is called and the processor is halted.

If any of the debugging console I/O operations are used in the application, the calling of the internal *fflush* and *fflush_console* functions can be un-commented to assure the internal console buffers get flushed.

```
/* call userPostMain() from appconfig.c */
      jsr userPostMain

/*   The fflush calls where removed because they added code */
/*   growth in cases where the user is not using any debugger IO. */
/*   Users should make these calls at the end of main if they use debugger IO */

/*    move.w #0,r2 */
/*    jsr    fflush              ; flush file IO */
/*    jsr    fflush_console ; flush console IO */

/*    end of program; halt CPU */

      nop
      debughlt
      stop
}
```

Chapter 3
Directory Structure

This section describes the directory structure of the Freescale DSP56800E_Quick_Start tool, located in the **<...>\Freescale\DSP56800E_Quick_Start r2.3** directory. Note that the root directory of the DSP56800E_Quick_Start may be changed during installation by the user. In general, the DSP56800E_Quick_Start software is organized by supported devices as it is explained in this chapter.

3.1 Root Directory

The root, or main, directory is organized as shown in Figure 3-1.

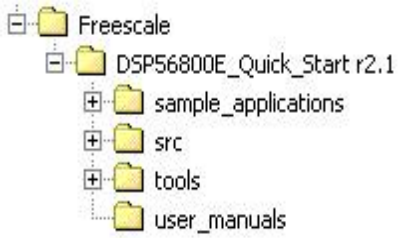


Figure 3-1. Root Directory Structure

Where:

- ***sample_applications*** contains simple application examples to demonstrate the usage of the DSP56800E_Quick_Start tool as well as the use of device or on-chip peripherals; see also Section 3.2.
- ***src*** contains the C source files; see also Section 3.4.
- ***stationery*** contains the templates for the newly created projects. Note: this directory is also installed directly into the CodeWarrior development tool, if the path to this tool was specified by user.
- ***tools*** contains the Graphical Configuration Tool, its configuration files, device data sheets and peripheral user manuals which can be opened directly from the GCT.
- ***user_manuals*** contains the DSP56800E_Quick_Start User's Manual and other documentation.

3.2 Sample Applications Directory

This directory contains simple application examples to demonstrate usage of the DSP56800E_Quick_Start tool as well as the usage of device or on-chip peripherals. The structure of the ***sample_applications*** directory is illustrated in Figure 3-2.

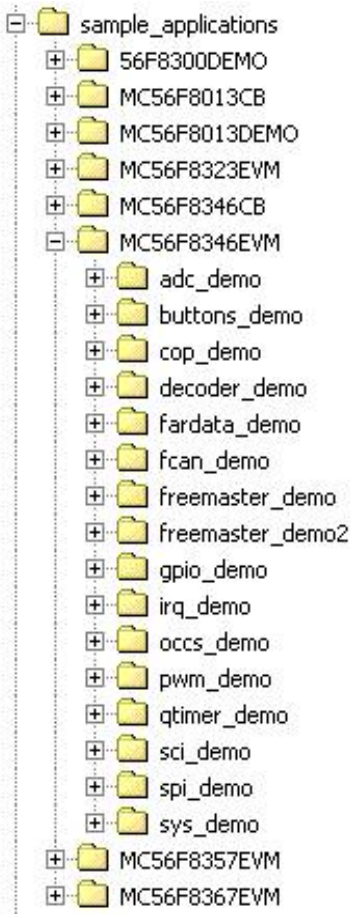


Figure 3-2. Sample Applications Directory Structure

The sample applications reside on the directory corresponding to the target hardware - Freescale EVM boards.

Note: the individual application directories are further structured with project specific folders which hold the configuration files, the project build files and the CodeWarrior private data files.

3.3 Tools Directory

The ***tools*** directory contains the Graphical Configuration Tool executable application, the needed *.dll* libraries and the help files.

3.4 Src Directory

The ***src*** or "source" directory is intended to hold all source files. Its structure is shown in Figure 3-3. The ***src*** directory is further divided into the following subdirectories:

- ***algorithms*** (optional) can contain the distributed algorithms and the user algorithms
- ***MC56F8xxx*** is the directory specific for each of the supported devices. The subdirectory ***peripheral*** contains the source code for all on-chip peripheral drivers and the ***system*** subdirectory contains the device specific source files
- ***include*** contains the common DSP56800E_Quick_Start header files, which define APIs and the implementation of generally used macro's
- ***support*** contains other common DSP56800E_Quick_Start source files. The subdirectory ***freemaster*** contains the source files to enable the FreeMASTER operation on the target board application. The other two directories, ***compat*** and ***pc_master***, are there because of compatibility with older DSP56800E_Quick_Start releases.

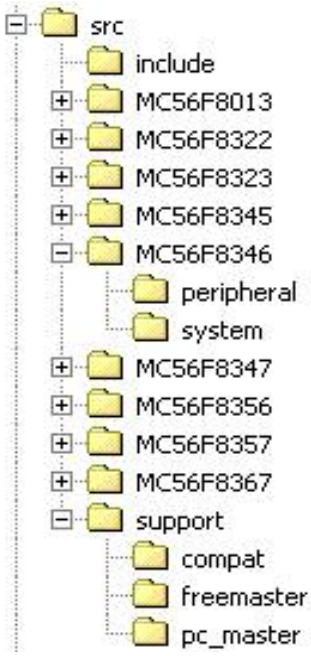


Figure 3-3. Src Directory Structure

3.5 Stationery Directory

The ***stationery*** directory contains the templates for the newly created DSP56800E_Quick_Start projects. This directory is also copied into the CodeWarrior development tool directory if a proper path was specified by the user. This directory may be needed to be copied manually into other CodeWarrior versions present on the host computer or into the CodeWarrior versions installed after the DSP56800E_Quick_Start was installed. See the detailed guide in Section 1.2.2.1.

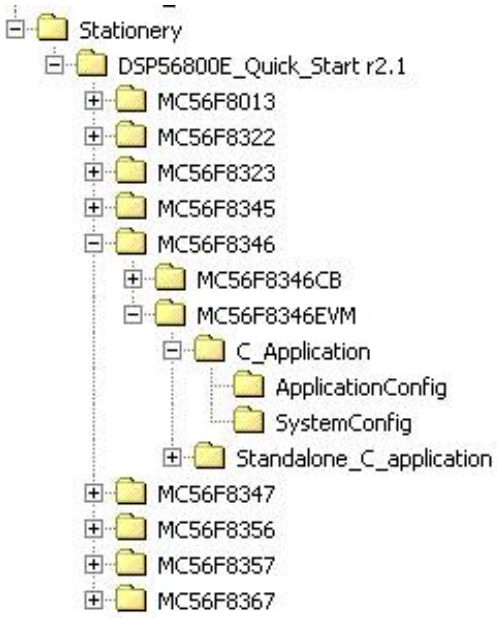


Figure 3-4. Stationery Directory Structure

The device specific subdirectories ***MC56F8xxx*** are covered by the DSP56800E_Quick_Start directory with the release version number. The device specific subdirectory contains all needed support files for the proper memory, system and application configuration and initialization.

3.6 User_manuals Directory

The ***user_manuals*** directory contains this DSP56800E_Quick_Start User's Manual as well as other relevant documentation.

Chapter 4
Developing Software

This chapter describes in detail how to develop the applications using the DSP56800E_Quick_Start. It describes how to create new applications using the DSP56800E_Quick_Start tool, how to initialize on-chip peripheral modules, how to access them in run-time by application code and an application configuration by an application specific configuration file *appconfig.h*. At this point it is assumed that CodeWarrior Development Tools and DSP56800E_Quick_Start are successfully installed and running (see Section 1.2.1 and Section 1.2.2 if you need information about installation of these tools).

4.1 Creating a new project

To create a new project based on the DSP56800E_Quick_Start project templates (stationery), perform the following steps:

1) Launch CodeWarrior IDE from the **Start->Programs->CodeWarrior** menu.

2) Choose **File->New** command.

3) Click the **Project** tab and select **DSP56800E_Quick_Start r2.3 Stationery** project type.

4) Type the project name (with a *.mcp* extension) and set the location for the new project.

5) Click **OK**.

6) Select the project stationery from the **New Project** window. This step includes selecting the type of processor (e.g. MC56F8346).

7) Click **OK**.

Upon completing all these actions the project window is displayed. The project window contains the predefined file groups:

- **Dependencies** - contains the following file subgroups:
  - **ApplicationConfig** - contains the *appconfig.h* header file.
  - **SystemConfig** - contains *boot.asm*, *startup.c*, *appconfig.c*, *vectors.c* and *arch.c* files.
  - **LinkerFiles** - contains the target specific linker command files *LDM_ExtRam.cmd*, *LDM_IntRam.cmd*, *LDM_xFlash*, *LDM_pFlash.cmd, SDM_ExtRam.cmd, SDM_pFlash.cmd, SDM_xFlash.cmd.*
  - **Lib** - contains the CodeWarrior standard libraries.
- **Drivers**
  - **Peripherals** - contains chip specific driver source files.

- **Support**
  - **FreeMASTER** - contains FreeMASTER software support files *freemaster.h* and *freemaster_xxx.c*.
- **Include**- contains header files for the driver source files.

Now, you can start writing your code in the C source file *main.c* and to configure the on-chip peripherals into the include file *appconfig.h,* either manually or by using the Graphical Configuration Tool (GCT). See Chapter 7 for GCT usage.

4.2 On-chip peripheral initialization

The DSP56800E_Quick_Start provides a very effective mechanism on how to initialize statically all on-chip peripherals. The static configuration of on-chip peripherals is provided by the application specific configuration file *appconfig.h*, in cooperation with the *ioctl* driver commands xx_INIT (xx is the peripheral prefix used in all *ioctl* commands). These commands are for example QT_INIT for Quad Timer, PWM_INIT for Pulse Width Modulator, etc.

The configuration file *appconfig.h* is used to define the configuration items, which determine the configuration of the on-chip peripheral. Each configuration item corresponds to one register of the respective on-chip peripheral. The defined configuration items are written to peripheral registers by the *ioctl* driver commands xx_INIT, which are called by the user somewhere in the initialization code of the application.

The step by step procedure to statically initialize the on-chip peripheral, using the DSP56800E_Quick_Start, is the following:

1. Define configuration items (register values) in the configuration file *appconfig.h*. You can edit the *appconfig.h* file manually or with the Graphical Configuration Tool (GCT). For more information on the GCT, refer to Chapter 7, “Graphical Configuration Tool,” and predefined names of all configuration items can be found in Chapter 5, “On-chip Drivers,” .
2. Initialize the selected on-chip peripheral by calling the xx_INIT *ioctl* command (xx is the peripheral prefix, for example QT_INIT for Quad Timer, PWM_INIT for Pulse Width Modulator etc.)

The DSP56800E_Quick_Start also enables to dynamically initialize on-chip peripherals. The dynamic configuration is fully supported by the *ioctl* commands. See Chapter 5, “On-chip Drivers,” where all *ioctl* commands are described.

Tip: If you are editing the configuration file *appconfig.h* manually, you can copy the template of all configuration items, intended for the *appconfig.h* file from the peripheral module header file *<name_of_driver>.h* (e.g. *intc.h* - Interrupt Controller driver include file, *pwm.h* - Pulse Width Modulation driver include file, etc.). See Example 4-1 where this template for timer/counter, extracted from the include file *qtimer.h*, is shown.

Example 4-1. Configuration items for timer/counter - extract from the driver header file

```
/*************************************************************************
        Defines for appconfig.h for each timer/counter
*************************************************************************

      void qtxxISR(void);
      #define INT_VECTOR_ADDR_yy qtxxISR
      #define INT_PRIORITY_LEVEL_yy one of INTC_DISABLED, INTC_LEVEL0,
                  INTC_LEVEL1, INTC_LEVEL2 or INTC_LEVEL3

      #define QT_xx_CTRL_INIT  0x0000
      #define QT_xx_SCR_INIT   0x0000
      #define QT_xx_CMP1_INIT  0x0000
      #define QT_xx_CMP2_INIT  0x0000
      #define QT_xx_LOAD_INIT 0x0000
      #define QT_xx_CNTR_INIT  0x0000

            where:
            xx is timer/counter:
                  D0,D1,D2,D3,C0,C1,C2,C3,B0,B1,B2,B3,A0,A1,A2,A3

            yy is corresponding interrupt vector
                  e.g. on MC56F8346:
                        D: 52,53,54,55
                        C: 56,57,58,59
                        B: 60,61,62,63
                        A: 64,65,66,67
*/
```

4.3 On-chip drivers - interface description

The DSP56800E_Quick_Start includes a set of on-chip drivers which are used to initialize, to configure and to access the on-chip peripherals. The on-chip drivers provide a C language Application Programming Interface (API) to the peripheral module (see Figure 4-1). This interface is common for all input/output operations.

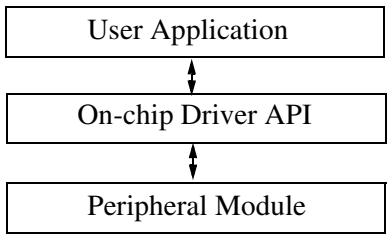


Figure 4-1. User Interface

This interface provides the following API statements:

ioctl - to initialize and to access peripheral module

read - e.g. to receive data

write - e.g. to transmit data

The philosophy of all input/output operations resides in these three statements (commands). These commands provide better code portability between the processors from the same family, where the base addresses of the peripheral modules are different. Therefore it is preferred to use

ioctl, *read* and *write* commands instead of the direct access to the peripheral module registers. The direct access to the peripheral registers is performed by *periphMemWrite*, *periphMemRead* and other predefined macros described in Section 2.4.2.

4.3.1 ioctl()

The **ioctl** command is used to initialize a peripheral module (see Section 4.2) and to access a peripheral module. Use of the *ioctl* command provides a very efficient and easy way to access a peripheral module. It increases the code portability and readability and, thus, decrease the number of bugs in the developed code.

The general syntax of the **ioctl** command is as follows:

```
ioctl(peripheral_module_identifier, command, command_specific_parameter);
```

or (if *ioctl* command returns a value):

```
var = ioctl(peripheral_module_identifier, command, command_specific_parameter);
```

Where:

- *Peripheral_module_identifier* parameter is the base address of the peripheral module. Instead of passing the raw base address (e.g. 0xF2D0) you can use the predefined symbolic constants OCCS, QTIMER, INTC etc.
- *Command* parameter specifies the action, which will be performed on the peripheral module. The list of all commands available can be found in Chapter 5, “On-chip Drivers,” .
- *Command_specific_parameter* parameter specifies other data required to execute the command.

Example 4-2. Using ioctl

```
ioctl(GPIO_B, GPIO_SET_PIN, BIT_1 | BIT_2);
ioctl(ADC_A, ADC_START, NULL);
ioctl(QTIMER_D1, QT_CLEAR_FLAG, QT_COMPARE_FLAG);
```

This example shows a miscellaneous ioctl commands. Note, the parameters are: the first one specifies the peripheral module (GPIO_B - General Purpose Input Output B, ADC_A - Analog to Digital Converter A, QTIMER_D1 - timer/counter D1), the second one is the command (GPIO_SET_PIN - to set pin, ADC_START - to start A/D conversion, QT_CLEAR_FLAG - to clear flag) and the third one is the command specific parameter (BIT_1 | BIT_2 - to specify that bits 1 and 2 will be set, NULL - no parameter is used, QT_COMPARE_FLAG - to clear timer compare flag).

See Chapter 5, “On-chip Drivers,” where all *ioctl* commands and their detailed descriptions can be found.

Tip: To see all available ioctl commands and their parameters from within CodeWarrior IDE, just open the appropriate <*name_of_driver*>.*h* include file (e.g. *intc.h* - Interrupt Controller driver include file, *pwm.h* - Pulse Width Modulation driver include file, etc.) and, at the beginning of the file, there is a list of all implemented *ioctl* commands.

4.3.2 read()

The **read**() function reads a specified number of words from the SCI or SPI module to an user allocated buffer. This function can operate in Blocking mode (it waits till end of operation), NonBlocking mode (it exits from the function immediately and the end of operation can be checked by testing the status word) or in Buffered mode (characters are copied from the internal circular buffer).

The syntax of **read**() function call is as follows:

```
read(peripheral_module_identifier, mode_identifier, buffer_pointer, number_of_words);
```

Where:

- *Peripheral_module_identifier* parameter is the same as in *ioctl()*, i.e. the base address of the peripheral module, which can be either SCI or SPI (because only these two peripheral modules can transmit/receive data). The predefined symbolic constants SCI or SPI can be used.
- *Mode_identifier* parameter specifies the mode of operation, which can be BLOCKING, NON_BLOCKING. or BUFFERED.
- *Buffer_pointer* and *number_of_words* parameters specifies the buffer pointer and the number of words, which will be read.

See Section 5.15.3.39 and Section 5.16.3.38 where the **read**() function is described.

4.3.3 write()

The **write**() function writes a specified number of words to the SCI or SPI module from the user allocated buffer. This function can operate in Blocking mode (it waits for end of operation), NonBlocking mode (it exits from the function immediately and the end of operation can be checked by testing the status word) or in Buffered mode (characters are copied into the internal circular buffer).

The syntax of the **write**() function call is as follows:

```
write(peripheral_module_identifier, mode_identifier, buffer_pointer,
            number_of_words);
```

Where:

- *Peripheral_module_identifier* parameter is the same as in *ioctl()*, i.e. the base address of the peripheral module, which can be either SCI or SPI (because only these two peripheral modules can transmit/receive data). The predefined symbolic constants SCI or SPI can be used.
- *Mode_identifier* parameter specifies the mode of operation, which can be BLOCKING, NON_BLOCKING. or BUFFERED.
- *Buffer_pointer* and *number_of_words* parameters specifies the buffer pointer and the number of words which will be transmitted.

See Section 5.15.3.40 and Section 5.16.3.39 where the **write**() function is described.

4.4 Interrupts and Interrupt Service Routines

Handling interrupts using the DSP56800E_Quick_Start is described in detail in Section 2.5. This section contains also a practical guide on how to write a user interrupt service routine.

4.5 *appconfig.h* file

The *appconfig.h* (see the full listing in Example 4-3) include file is the application specific configuration file. It is used to define configuration items and the addresses of the user interrupt service routines (ISRs). The defined configuration items are then used by the *ioctl()* commands xx_INIT to initialize statically the xx peripheral module (e.g. OCCS_INIT to initialize On-chip Clock Synthesis module, QT_INIT to initialize quad timer/counter, etc.). The defined addresses of ISRs are used to install the ISR into the desired interrupt vector, at compilation time.

See Section 4.2 to find out how to initialize on-chip peripheral modules using the *appconfig.h* file and the respective ioctl xx_INIT command.

See Section 2.5.2, “Configuring Interrupts,” for information on installing ISRs and defining the interrupt priorities through the *appconfig.h* file.

Example 4-3. *appconfig.h* from sample application

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Mon, 26/Sep/2005, 11:28:36
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02010004L

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 4 MHz
    VCO frequency: 240 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0081
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
```

```
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output Mode: Off: Tristated
.*/
#define SIM_GPS_INIT                        0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                    0x0005
#define SEMI_CSBAR1_INIT                    0x0005
#define SEMI_CSBAR2_INIT                    0x0005
#define SEMI_CSOR0_INIT                     0x1FC3
#define SEMI_CSOR1_INIT                     0x1BA3
#define SEMI_CSOR2_INIT                     0x1DA3

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                      0x0000
#define INT_VECTOR_ADDR_78                  pwm_Reload_A_Callback
#define INT_PRIORITY_LEVEL_78               INTC_LEVEL1

/*.
    PWM_A Configuration
-----------------------------------------
    PWM module operation: Disabled
    Prescaler: /8 , PWM clock period: 2 us
    PWM frequency: 7.62963 Hz , Period: 131.06801 ms
    PWM Deadtime: n/a
    PWM Reload Frequency: Every 10 opportunity
    Alignment: Center
```

```
    Debug Mode Operation: Stop
    Wait Mode Operation: Stop
    Deadtime Correction Method: Manual correction (no correction)
    Half Cycle Reload: Disabled
    Value Register Load Hardware Acceleration: Disabled
    Output Pad Enable: Enabled
    Load OK: Yes
    Swap & Mask: DSP56F80X compatible
    Write Protection: No
    Top Polarity   : Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Bottom Polarity: Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Chann. coupling: Channels 4-5: Complementary
                     Channels 2-3: Complementary
                     Channels 0-1: Complementary
    Clearing Mode:   Fault 0 pin: Manual
                     Fault 1 pin: Manual
                     Fault 2 pin: Manual
                     Fault 3 pin: Manual
    Asymmetric operation Channels 0-1: Off (Correction Method used only)
    Asymmetric operation Channels 2-3: Off (Correction Method used only)
    Asymmetric operation Channels 4-5: Off (Correction Method used only)
.*/
#define PWM_A_PMCTL_INIT                  0x90E2
#define PWM_A_PMOUT_INIT                  0x8000
#define PWM_A_PWMCM_INIT                  0x7FFF
#define PWM_A_PMDEADTM_INIT               0x0FFF
#define PWM_A_PMDISMAP1_INIT              0x0000
#define PWM_A_PMDISMAP2_INIT              0x0000

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  9: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin 10: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
.*/
#define GPIO_C_DDR_INIT                   0x000F
#define GPIO_C_PER_INIT                   0x00F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
```

```
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                 0x1FC1


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Chapter 5
On-chip Drivers

One of the DSP56800E_Quick_Start tool strengths is that it provides a high degree of architectural and hardware independence for the application code. This portability is achieved by the modular design of DSP56800E_Quick_Start, which in this case, isolates all chip-specific functionality into a set of defined, tested and documented Application Programming Interface (API).

This chapter describes the API for on-chip drivers, forming the interface between hardware and application software. The source code - implementation can be found at ***<...>\src\MC56F8xxx\peripheral*** of the DSP56800E_Quick_Start. It defines the API by identifying all public interface functions-commands and data structures.

The DSP56800E_Quick_Start on-chip driver's API is implemented as a *low level device driver interface*. The low level device driver interface was chosen mainly for its efficiency and also because it enables the utilization of the whole hardware functionality. Another reason is the non standardized approach, on how to use most of the on-chip peripheral modules. The portability of the low level device driver interface is not influenced so much by the lower abstraction level, but mainly by the capability of the peripheral module hardware. It means that, the portability is ensured between devices, which involves the same or a very similar implementation of the peripheral module hardware. In case of quite different peripheral modules on target devices, the portability is much lower. Nevertheless, in such a case, the overall application might be built differently, just to reflect the hardware capability.

The general introductory description of on-chip drivers can be found in Section 4.3. Here are some more ideas or hints.

On-chip drivers usage considerations:

- ***Peripheral module hardware and functionality knowledge.***
  The only efficient and, in some cases, safe usage of the on-chip peripheral module is based on the user knowledge that the user has about the module itself. A comprehensive description can be found in the **MC56F8300 Peripheral User Manual**, **56F8000 Peripheral Reference Manual** and in various Freescale/Motorola Application Notes (AN's). The way in which the on-chip driver's API is designed takes advantage of the whole hardware capability. The self-explaining names of the driver commands will help the users to find the desired hardware feature.
- ***On-chip driver commands implemented as macros.***
  Almost all commands are implemented as efficient C function-like macros. The exceptions are the initialization commands and the read/write commands of the SPI and the SCI, which are implemented as regular functions. This reality is documented in each detailed description of the command.
  The efficiency is not only the reason for an implementation as macros. The other advantage is an easy use within the interrupt service routine, where the unwanted overhead (i.e. jump/return to/from function plus context store/restore) is thus eliminated. Further, the consistent implementation of

commands takes care of the used instructions, mainly due to the proper usage of the read-modify-write instructions. These uninterruptible instructions are essential from the safety point of view, when accessing the control and peripheral registers.

- ***Efficient use of the driver commands.***
  The general form of the driver command is the following:

```
ioctl(peripheral_module_identifier,command,command_specific_parameter);
```

Where, the *Peripheral_module_identifier* parameter is the base address of the peripheral module. The predefined symbolic constants, like OCCS, PWM_A, INTC, DEC_0 etc., should be used. The *Command* parameter specifies the action, which will be performed on the peripheral module. It represents the command name as it is implemented for each on-chip driver.
The *Command_specific_parameter* parameter specifies other data required to execute the command. Generally speaking, it can be a pointer to the structure, the NULL value or a variable-value in dependency with the specific command. If the required parameter is variable-value, it is recommended to use a constant value if possible, because it influences the efficiency of the resulting code. The efficiency is illustrated by the following examples:

```
ioctl( PWM_A, PWM_SET_MODULO, 0x30ff ); //constant used
```
results in:
```
       move.l   0xf145,R0              or:
       move.w   #12543,X:(R0)          move.w   #12543,X:0xf145
```
while the following code sequence
```
val = 32767;
...
ioctl( PWM_A, PWM_SET_MODULO, val ); //variable used
```
results in:
```
       move.w   #32767,X:0x003012
       ...
       move.l   0x003012,R1
       move.l   0xf145,R0              or:
       move.w   X:(R1),A               move.w   #32767,A
       move.w   A1,X:(R0)              move.w   A1,X:0xf145
```
Another possibility is to use the predefined symbolic constants to express the desired action, like:
```
ioctl( PWM_A, PWM_RELOAD_INT, PWM_ENABLE );
```
which results in:
```
       move.l   0xf140,R0              or:
       bfset   #0x20,X:(R0)            bfset   #0x20,X:0xf140
```
So, the predefined symbols or constants should be used whenever possible.

Note: Some macros expand just to a single assembly instruction (as illustrated above). Some other macros expand to more assembly instructions, e.g. the different mode setting where it is necessary to clear the previous setting and then, to set the new mode. This is illustrated by the following example:

```
ioctl( PWM_A, PWM_SET_PRESCALER, PWM_PRESCALER_DIV_2 );

       move.l   0xf140,R0              or:
       bfclr   #0xc0,X:(R0)            bfclr   #0xc0,X:0xf140
       bfset   #0x40,X:(R0)            bfset   #0x40,X:0xf140
```

Note that the generated code depends on the selected compiler optimizations and also on the rest of the source code and on the selected target.

There can be even longer commands. These commands incorporate some higher functionality than only a simple access to the peripheral registers. An example can be the commands, which perform the mathematical calculations for data scaling to fit the results into the desired data range, like recounting of the PWM duty cycle in percentage of the actual value to be written to the PWM Value register.

Example 5-1. Implementation details

Figure 5-1 is intended to illustrate the macro expansion process. The corresponding items are highlighted by the same color through this macro expansion example.

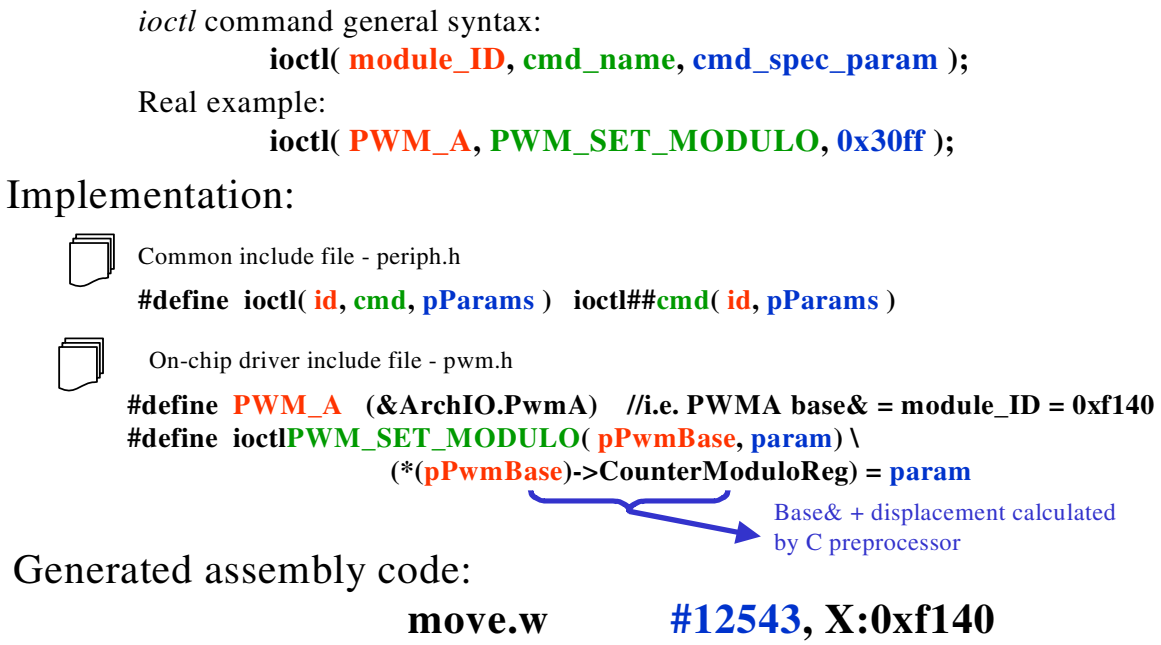

ioctl command general syntax:
ioctl(module_ID, cmd_name, cmd_spec_param);
Real example:
ioctl(PWM_A, PWM_SET_MODULO, 0x30ff);
Implementation:
Common include file - periph.h
#define ioctl(id, cmd, pParams) ioctl##cmd(id, pParams)
On-chip driver include file - pwm.h
#define PWM_A (&ArchIO.PwmA) //i.e. PWMA base& = module_ID = 0xf140
#define ioctlPWM_SET_MODULO(pPwmBase, param) \
(*(pPwmBase)->CounterModuloReg) = param
Base& + displacement calculated
by C preprocessor
Generated assembly code:
move.w #12543, X:0xf140

Figure 5-1. Macro Expansion Process

5.1 OCCS Driver

This section describes the DSP56800E_Quick_Start API for the MC56F83xx and MC56F80xx OCCS on-chip module. The functionality of the OCCS module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.1.1 Introduction

The On-Chip Clock Synthesis (OCCS) module is available on all 56F800E chips and provides the 2X system clock frequency to the system integration module (SIM), which uses it to generate the various chip clocks. The main blocks of OCCS are frequency Prescaler, Postscaler, PLL, clock multiplexer and lock detector. The main OCCS module features are:

- possibility to use the PLL to multiply the externally generated clock
- PLL loss-of-lock detector with interrupt capability
- Prescaler to divide the input frequency (on MC56F80xx is prescaler fixed to 1 by hardware)
- Postscaler to divide the PLL output frequency

Note that the OCCS module on certain devices is equipped with internal relaxation oscillator, so no external clock source is needed. The OCCS module on MC56F80xx devices is capable to generate 3X system clock, called High Speed Peripheral Clock (HS_PERF), to selected peripheral modules.

There are differences in the way the OCCS module is controlled on different 56F800E sub-families. The MC56F83xx, MC56F801x and MC56F802x/3x devices each uses a slightly different PLL modules. In any case, it is recommended to configure the OCCS statically in the Graphical Configuration Tool and let the DSP56800E_Quick_Start standard start-up code to apply the configuration.

The following sections describe the OCCS driver software which provides the low level run-time API to the OCCS hardware.

5.1.2 Quick Reference

This section defines the terms and formulas used later in Section 5.1.3.

Table 5-1. OCCS Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
OCCS (PLL_BASE)	0xF160	0xF0F0	0xF130	0xF2D0

5.1.2.1 OCCS frequency calculation

For exact frequency formulas, please see the OCCS documentation in the Peripheral Reference Manual for each sub-family. Different Peripheral Manuals exist for MC56F83xx, MC56F801x, MC56F802x/3x and MC56F800x sub-families.

5.1.2.2 API Definition

The following header files are needed in order to use the OCCS device driver:

Required Header File(s):

```
#include "qs.h"
#include "occs.h"
```

The following information may be found in the header file *occs.h*.

Public Data Structure(s):

none

General-Use Macros:

```
#define OCCS_MSTROSC_FREQ   /* MSTR_OSC frequency calculated from
                               OCCS configuration in appconfig.h.
                               This frequency is supplied as an
                               input to PLL */

#define OCCS_CORE_FREQ      /* core frequency calculated from
                               OCCS_PLLCR_INIT and OCCS_PLLDB_INIT
                               and EXTCLK values */
```

5.1.2.3 Configuration Items

This section summarizes the symbols used in macro definitions for the static OCCS module configuration. These symbols are intended for the project specific configuration file *appconfig.h* and are applied by the startup code.

Table 5-2. OCCS Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
EXTCLK	UWord32	Specifies the external clock frequency in Hz. This value is used to calculate and define the OCCS_CORE_FREQ macro specifying the real core clock frequency. The value is based on the other *appconfig.h* symbols.	The OCCS_CORE_FREQ value is not available for use.
only on MC56F800x: EXTAL	UWord32	Specifies the Crystal clock frequency in Hz. This value is used to calculate and define the OCCS_CORE_FREQ macro specifying the real core clock frequency if crystal owscillator is sellected as source of clock. The value is based on the other *appconfig.h* symbols.	The OCCS_CORE_FREQ value is not available for use.

Table 5-2. OCCS Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
OCCS_PLLCR_INIT	UWord16	Initial value of the PLL Control Register (PLLCR). PLL is set up by the startup code if Postscaler output is to be used as ZCLK.	Reset value used. PLL is not set up by the startup code.
OCCS_PLLDB_INIT	UWord16	Initial value of the PLL Divide-by Register (PLLDB).	Reset value used.
OCCS_OSCTL_INIT	UWord16	Initial value of the Oscillator Control Register (OSCTL).	Reset value used.
only on MC56F802x/3x: OCCS_PROT_INIT	UWord16	Initial value of the Protection Register (PROT).	Reset value used.
OCCS_REQUIRED_LOCK_MODE	UWord16	Specifies the PLL lock state in which the startup code considers the PLL stable. The valid values are: 0x20 - fine (LCK0 bit in PLLSR) 0x40 - coarse (LCK1 bit in PLLSR)	0x40 used.
Devices equipped with internal relaxation oscillator:			
USE_FACTORY_TRIM	zero / non-zero	When non-zero, the startup code initializes the Oscillator Trimming register with the factory-measured value.	Same as false. Trimming does not occur.
RXOSC_RETRIM_CLK:	UWord32	When USE_FACTORY_TRIM is non-zero, this macro may specify required oscillator frequency. The startup code then tries to achieve this frequency by calculating proper trimming value.	The value of 8000000 is used. The default factory-measured Trimming Value is used, making the resulting 8 MHz oscillator clock.

5.1.2.4 API Specification

This section briefly describes the API macros and functions.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by the following form:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
```

Description: The *ioctl* call "changes" OCCS device modes or accesses the OCCS register(s). Keep in mind that *ioctl* is treated as macro and that in result it is mostly compiled to an optimal inline code.

Arguments:

Table 5-3. OCCS Driver Arguments - ioctl

pModuleBase	in	OCCS module identifier. Use OCCS.
cmd	in	Command names found in *occs.h.* See Table 5-4.
pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-4. ioctl commands

Cmd	param, pParam	Return	Description
OCCS_INIT	NULL	None	Does nothing. Maintained only for backward compatibility. Note that PLL is initialized early during the startup code.
OCCS_SET_CORE_CLOCK	UWord16	None	Sets DSC core frequency according to values passed as pParam parameter.
OCCS_SET_POSTSCALER	OCCS_CLOCK_OUT_DIVIDE_BY_1 / OCCS_CLOCK_OUT_DIVIDE_BY_2 / OCCS_CLOCK_OUT_DIVIDE_BY_4 / OCCS_CLOCK_OUT_DIVIDE_BY_8 **and on MC56F80xx additional:** OCCS_CLOCK_OUT_DIVIDE_BY_16 / OCCS_CLOCK_OUT_DIVIDE_BY_32 **and on MC56F800x additional:** OCCS_CLOCK_OUT_DIVIDE_BY_64 / OCCS_CLOCK_OUT_DIVIDE_BY_128 / OCCS_CLOCK_OUT_DIVIDE_BY_256	None	Sets Postscaler value.

Table 5-4. ioctl commands (Continued)

Cmd	param, pParam	Return	Description
only on MC56F83xx: OCCS_SET_PRESCALER	OCCS_CLOCK_IN_DIVIDE_BY_1 / OCCS_CLOCK_IN_DIVIDE_BY_2 / OCCS_CLOCK_IN_DIVIDE_BY_4 / OCCS_CLOCK_IN_DIVIDE_BY_8	None	Sets Prescaler value.
only on MC56F83xx: OCCS_SET_DIVIDE_BY	UWord16	None	Sets Divide-By value of the PLL (frequency multiplication factor).
OCCS_INT_ENABLE	(OCCS_LOL1_INT_ANY_EDGE / OCCS_LOL1_INT_FALLING_EDGE / OCCS_LOL1_INT_RISING_EDGE) | (OCCS_LOL0_INT_ANY_EDGE / OCCS_LOL0_INT_FALLING_EDGE / OCCS_LOL0_INT_RISING_EDGE) | OCCS_LOSS_OF_CLOCK_INT	None	Enables interrupts for different lock mode changes. LOL0 represents the LCK0 bit in PLLSR register, LOL1 represents the LCK1 bit.
OCCS_INT_DISABLE	OCCS_LOL1_INT | OCCS_LOL0_INT | OCCS_LOSS_OF_CLOCK_INT	None	Disables selected interrupts.
OCCS_LOCK_DETECTOR	OCCS_ENABLE / OCCS_DISABLE	None	Enables/disables lock detector.
only on MC56F83xx and MC56F801x: OCCS_TURN_OFF_CHARGE_PUMP	NULL	None	Sets Charge Pump Tri-state bit (CHPMPTRI), bit 6 in PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).
OCCS_SET_ZCLOCK_SOURCE	**on MC56F83xx:** OCCS_POSTSCALER_OUTPUT / OCCS_PRESCALER_OUTPUT **or on MC56F801x and MC56F802x/3x:** OCCS_POSTSCALER_OUTPUT / OCCS_MSTR_OSC_OUTPUT **or on MC56F800x:** OCCS_PLL_OSC_OUTPUT / OCCS_MSTR_OSC_OUTPUT	None	Selects the clock provided as the source to the SIM module.
OCCS_GET_ZCLOCK_SOURCE	NULL	UWord16	Returns the currently selected clock source.
OCCS_READ_FLAG	OCCS_STATUS_LOCK_LOST_INT1 | OCCS_STATUS_LOCK_LOST_INT0 | OCCS_STATUS_CLOCK_LOST | OCCS_STATUS_LOCK_1 | OCCS_STATUS_LOCK_0 | OCCS_STATUS_POWER_DOWN | OCCS_STATUS_ZCLOCK **and on MC56F800x additional:** OCCS_STATUS_CRYSTAL_READY	UWord16	Returns status of selected flags in PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).
OCCS_CLEAR_FLAG	OCCS_STATUS_LOCK_LOST_INT1 | OCCS_STATUS_LOCK_LOST_INT0 | OCCS_STATUS_CLOCK_LOST	None	Clears selected flags in PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

Table 5-4. ioctl commands (Continued)

Cmd	param, pParam	Return	Description
OCCS_SET_LORTP	UWord16	None	Sets Loss of Reference Timer Period (LORTP).
OCCS_GET_IPBUS_FREQ	oscillator frequency [Hz] (UInt32)	Clock freq. [Hz] (UInt32)	Returns current clock frequency. The returned frequency is calculated according to settings in OCCS module registers.
OCCS_WRITE_CONTROL_REG	UWord16	None	Writes 16-bit value to the PLL Control Register (PLLSR on MC56F83xx, STAT on MC56F80xx).
OCCS_WRITE_DIVIDE_BY_REG	UWord16	None	Writes 16-bit value to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).
OCCS_WRITE_OSC_CONTROL_REG	UWord16	None	Writes 16-bit value to the Oscillator Control Register (OSCTL).
OCCS_READ_CONTROL_REG	NULL	UWord16	Returns the content of the PLL Control Register (PLLSR on MC56F83xx, STAT on MC56F80xx).
OCCS_READ_DIVIDE_BY_REG	NULL	UWord16	Returns the content of the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).
OCCS_READ_STATUS_REG	NULL	UWord16	Returns the content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).
OCCS_READ_OSC_CONTROL_REG	NULL	UWord16	Returns the content of the Oscillator Control Register (OSCTL).
only on MC56F83xx, MC56F802x/3x and MC56F800x: OCCS_POWER_MODE	OCCS_HIGH_POWER / OCCS_LOW_POWER	None	Controls the power usage of the crystal oscillator.
only on MC56F83xx and MC56F801x: OCCS_SHUTDOWN	UWord16	None	Shuts-down all system clocks.
only on devices w/ internal relax.osc.: OCCS_SET_PRESCALER_CLOCK	OCCS_INTERNAL_RELAX_OSC / OCCS_CRYSTAL_OSC	None	Set the prescaler clock source.
only on devices w/ internal relax.osc.: OCCS_INTERNAL_RELAX_OSC_ OPERATION	OCCS_ENABLE / OCCS_DISABLE **on MC56F80xx additional:** OCCS_STANDBY	None	Enables or powers-down the relaxation oscillator.

Table 5-4. ioctl commands (Continued)

Cmd	param, pParam	Return	Description
only on devices w/ internal relax.osc.: OCCS_ADJUST_RELAX_OSC_FREQ	UWord16 (0..1023)	None	Adjusts the relaxation oscillator frequency.
only on devices w/ internal relax.osc.: OCCS_TRIM_RELAX_OSC_8MHZ	NULL	None	Adjusts the relaxation oscillator frequency to 8MHz by factory settings.
only on MC56F802x/3x and MC56F800x: OCCS_DIRECT_CLOCK_MODE	OCCS_ENABLE / OCCS_DISABLE	None	Sets clock-in mode on XTAL pin (turns off the oscillator).
only on MC56F802x/3x andMC56F800x : OCCS_SELECT_EXT_CLOCK_ SOURCE	**on MC56F802x/3x:** OCCS_CLKIN_PRI / OCCS_CLKIN_ALT / OCCS_CLKIN_OSC **on MC56F800x:** OCCS_CLKIN_CLKIN / OCCS_CLKIN_EXTAL	None	Selects which clock-input pin will be used as a direct clock source. The GPIO pins must be configured accordingly.
only on MC56F802x/3x and MC56F800x: OCCS_WPROTECT_PLL_SETTINGS	OCCS_ENABLE / OCCS_ENABLE_PERMANENT / OCCS_DISABLE / OCCS_DISABLE_PERMANENT	None	Write-protect the PLL configuration bits (PLLPDN, LOCIE and LORTP bit-fields).
only on MC56F802x/3x and MC56F800x: OCCS_WPROTECT_OSC_SETTINGS	OCCS_ENABLE / OCCS_ENABLE_PERMANENT / OCCS_DISABLE / OCCS_DISABLE_PERMANENT	None	Write-protect the Oscillator configuration bits (OSCTL register and PRECS bit-field).
only on MC56F802x/3x and MC56F800x: OCCS_WPROTECT_CLK_SETTINGS	OCCS_ENABLE / OCCS_ENABLE_PERMANENT / OCCS_DISABLE / OCCS_DISABLE_PERMANENT	None	Write-protect the Frequency configuration bits (PLLCOD and ZSRCS bit-fields).
only on MC56F802x/3x and MC56F800x: OCCS_SET_CLOCK_CHECK	OCCS_ENABLE / OCCS_DISABLE	None	Start/Stop the clock checking function
only on MC56F802x/3x and MC56F800x: OCCS_TEST_CLOCK_CHECK	NULL	UWord16	Returns the number of ROSC cycles that have been counted since last reset counter.
only on MC56F802x/3x and MC56F800x: OCCS_READ_CLOCK_CHECK_ REFERENCE	NULL	UWord16	Returns the number of ROSC cycles that have been counted since last reset counter.
only on MC56F802x/3x and MC56F800x: OCCS_READ_CLOCK_CHECK_ TARGET	NULL	UWord16	Returns the namber of external clock cycles that have been counted since last reset counter.
only on MC56F800x: OCCS_SELECT_FREQ_RANGE	OCCS_32KHZ_CRYSTAL / OCCS__1MHZ_TO_16MHZ_CRYSTAL	none	Selects the frequency range of the crystal oscillator

5.1.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.1.3.1 OCCS_INIT - initialize OCCS module

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_INIT, NULL);
```

Arguments:

Table 5-5. OCCS_INIT ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: This command is maintained for backward compatibility only. Its current implementation is empty.

In previous releases for DSP56F800 (HawkV1) family, this command initialized the OCCS module according to the values from *appconfig.h* configuration file. In the MC56F800E (HawkV2) release, the OCCS module is set up already during the startup code.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_INIT* ioctl command is implemented as a function call.

5.1.3.2 OCCS_SET_CORE_CLOCK - set DSC core clock frequency

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_CORE_CLOCK,
           UWord16 param);
```

Arguments:

Table 5-6. OCCS_SET_CORE_CLOCK ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Content of the PLL Divide-By Register (PLLDB). Instead of passing raw hex value you can use a combination of these predefined constants: **on MC56F83xx:** OCCS_CLOCK_IN_DIVIDE_BY_x | OCCS_CLOCK_OUT_DIVIDE_BY_y | DivBy where: x - 1, 2, 4 or 8 (determines prescaler divide-by value) y - 1, 2, 4 or 8 (determines postscaler divide-by value) DivBy - Divide-By value in range 0 to 127 **on MC56F80xx:** OCCS_CLOCK_OUT_DIVIDE_BY_y where: y - 1, 2, 4, 8, 16 or 32(determines postscaler divide-by value) **and on MC56F800x:** y - 1, 2, 4, 8, 16, 32, 64, 128 or 256

Description: The *OCCS_SET_CORE_CLOCK* ioctl command configures the On-Chip Clock Synthesis (OCCS) module to the most frequently used mode, when the PLL block provides clock to the DSC core (ZCLOCK Source is set to Postscaler output). First the command sets the ZCLOCK Source to Prescaler output and turns the lock detector on. Then it writes the "param" value to the PLL Divide by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx) and it waits until the PLL is locked. Finally it switches the ZCLOCK Source to the Postscaler output. See Section 5.1.2.1, "OCCS frequency calculation," for reference on expressions which are used to calculate prescaler, postscaler and divide-by value, according to the desired DSC core frequency or the IPBus clock frequency.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_SET_CORE_CLOCK* ioctl command is implemented as a macro.

Example 5-2. OCCS_SET_CORE_CLOCK

```
ioctl(OCCS, OCCS_SET_CORE_CLOCK, OCCS_CLOCK_IN_DIVIDE_BY_1 |
                                 OCCS_CLOCK_OUT_DIVIDE_BY_1 |
                                 19);
```

This code sets the MC56F83xx core frequency to 80MHz (if external crystal frequency is 8MHz).

5.1.3.3 OCCS_SET_POSTSCALER - set postscaler

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_POSTSCALER,
           UWord16 param);
```

Arguments:

Table 5-7. OCCS_SET_POSTSCALER ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of these predefined constants: OCCS_CLOCK_OUT_DIVIDE_BY_1 / OCCS_CLOCK_OUT_DIVIDE_BY_2 / OCCS_CLOCK_OUT_DIVIDE_BY_4 / OCCS_CLOCK_OUT_DIVIDE_BY_8 **and on MC56F801x and MC56F802x/3x additional:** OCCS_CLOCK_OUT_DIVIDE_BY_16 / OCCS_CLOCK_OUT_DIVIDE_BY_32 **and on MC56F800x additional:** OCCS_CLOCK_OUT_DIVIDE_BY_64 / OCCS_CLOCK_OUT_DIVIDE_BY_128/ OCCS_CLOCK_OUT_DIVIDE_BY_256

Description: The *OCCS_SET_POSTSCALER* ioctl command sets the postscaler. The output clock can be divided by 1, 2, 4, 8 on 56F83xx or 1, 2, 4, 8, 16, 32 on MC56F801x and MC56F802x/3x or 1, 2, 4, 8, 16, 32, 64, 128, 256 on MC56F800x. This command writes to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx), bits PLLCOD.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_SET_POSTSCALER* ioctl command is implemented as a macro.

Example 5-3. OCCS_SET_POSTSCALER

```
ioctl(OCCS, OCCS_SET_POSTSCALER, OCCS_CLOCK_OUT_DIVIDE_BY_8);
```

This code sets postscaler to divide by 8.

5.1.3.4 OCCS_SET_PRESCALER - set prescaler

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-8. OCCS_SET_PRESCALER ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of these predefined constants: OCCS_CLOCK_IN_DIVIDE_BY_1 / OCCS_CLOCK_IN_DIVIDE_BY_2 / OCCS_CLOCK_IN_DIVIDE_BY_4 / OCCS_CLOCK_IN_DIVIDE_BY_8

Description: The *OCCS_SET_PRESCALER* ioctl command sets the 2-bit prescaler. The PLL input clock can be divided by 1, 2, 4 or 8. This command writes to the PLL Divide-by Register (PLLDB), Bits 9-8 (PLLCID).

Returns: None.

Range Issues: None.

Special Issues: Use this command only when the ZCLOCK Source is set to the prescaler output. This command is applicable only on MC56F83xx.

Design/Implementation: The *OCCS_SET_PRESCALER* ioctl command is implemented as a macro.

Example 5-4. OCCS_SET_PRESCALER

```
ioctl(OCCS, OCCS_SET_PRESCALER, OCCS_CLOCK_IN_DIVIDE_BY_1);
```

This code sets the prescaler to divide by 1.

5.1.3.5 OCCS_SET_DIVIDE_BY - set Divide-by value

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_DIVIDE_BY,
           UWord16 param);
```

Arguments:

Table 5-9. OCCS_SET_DIVIDE_BY ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Divide-by value. Permissible range is 0 to 127.

Description: The *OCCS_SET_DIVIDE_BY* ioctl command sets the Divide-by value. This command writes passed parameter param to the PLL Divide-by Register (PLLDB), Bits 6-0 (PLLDB).

Returns: None.

Range Issues: None.

Special Issues: Use this command only when the ZCLOCK Source is set to the prescaler output. This command is applicable only on MC56F83xx.

Design/Implementation: The *OCCS_SET_DIVIDE_BY* ioctl command is implemented as a macro.

Example 5-5. OCCS_SET_DIVIDE_BY

```
ioctl(OCCS, OCCS_SET_DIVIDE_BY, 19);
```

This code sets divide by value to 19.

5.1.3.6 OCCS_INT_ENABLE - enable OCCS interrupts

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-10. OCCS_INT_ENABLE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the OCCS interrupts and the edge sensitivity. Use these predefined constants: ((OCCS_LOL1_INT_ANY_EDGE / OCCS_LOL1_INT_FALLING_EDGE / OCCS_LOL1_INT_RISING_EDGE) | (OCCS_LOL0_INT_ANY_EDGE / OCCS_LOL0_INT_FALLING_EDGE / OCCS_LOL0_INT_RISING_EDGE) | OCCS_LOSS_OF_CLOCK_INT

Description: The *OCCS_INT_ENABLE* ioctl command enables OCCS interrupts. The OCCS interrupts are PLL Interrupt 1 (if bit LCK1 in the PLL Status Register PLLSR changes), PLL Interrupt 0 (if bit LCK0 in the PLL Status Register PLLSR changes) and Loss of Clock Interrupt (if the oscillator circuit output clock is lost). This command manipulates the PLLIE1 (Bits 15-14), PLLIE0 (Bits 13-12) and LOCIE (Bit 11) bits in the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).

Returns: None.

Range Issues: None.

Special Issues: PLL Interrupt 1 and PLL Interrupt 0 can be enabled only if they were disabled before the usage of this command. It is not possible to change the mode of enabled PLL Interrupt 1 or 0 (for example from any edge to falling edge).

Design/Implementation: The *OCCS_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-6. OCCS_INT_ENABLE

```
ioctl(OCCS, OCCS_INT_ENABLE, OCCS_LOL1_INT_ANY_EDGE |
                                 OCCS_LOSS_OF_CLOCK_INT);
```

This code enables PLL Interrupt 1 and Loss of Clock Interrupt. It is also specified that PLL Interrupt 1 is sensitive to any edge change of bit LCK1 in the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

5.1.3.7 OCCS_INT_DISABLE - disable OCCS interrupts

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-11. OCCS_INT_DISABLE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the OCCS interrupts. Use these predefined constants: OCCS_LOL1_INT | OCCS_LOL1_INT | OCCS_LOSS_OF_CLOCK_INT

Description: The *OCCS_INT_DISABLE* ioctl command disables OCCS interrupts. The OCCS interrupts are PLL Interrupt 1 (if bit LCK1 in the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx) changes), PLL Interrupt 0 (if bit LCK0 in the PLL Status Register PLLSR changes) and Loss of Clock Interrupt (if the oscillator circuit output clock is lost). This command clears corresponding bits in the PLL Control Register (PLLCR). These bits are PLLIE1 (Bits 15-14) when OCCS_INT1 is used as a parameter, PLLIE0 (Bits 13-12) when OCCS_INT0 is used as a parameter and Loss of Clock Interrupt Enable (LOCIE Bit 11) when OCCS_LOSS_OF_CLOCK_INT is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-7. OCCS_INT_DISABLE

```
ioctl(OCCS, OCCS_INT_DISABLE, OCCS_LOL1_INT | OCCS_LOL0_INT |
OCCS_LOSS_OF_CLOCK_INT);
```

This code disables all OCCS interrupts - PLL Interrupt 1, PLL Interrupt 0 and Loss of Clock Interrupt.

5.1.3.8 OCCS_LOCK_DETECTOR - enable/disable lock detector

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_LOCK_DETECTOR,
           UWord16 param);
```

Arguments:

Table 5-12. OCCS_LOCK_DETECTOR ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the desired action. Use one of these predefined constants: OCCS_ENABLE / OCCS_DISABLE

Description: The *OCCS_LOCK_DETECTOR* ioctl command enables/disables the lock detector. This command sets the LCKON bit (Bit 7) in the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx) when OCCS_ENABLE is used as a parameter. This command clears the above mentioned bit in case of OCCS_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_LOCK_DETECTOR* ioctl command is implemented as a macro.

Example 5-8. OCCS_LOCK_DETECTOR

```
ioctl(OCCS, OCCS_LOCK_DETECTOR, OCCS_ENABLE);
```

This code enables lock detector.

5.1.3.9 OCCS_TURN_OFF_CHARGE_PUMP - turn off charge pump

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_TURN_OFF_CHARGE_PUMP,
           NULL);
```

Arguments:

Table 5-13. OCCS_TURN_OFF_CHARGE_PUMP ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_TURN_OFF_CHARGE_PUMP* ioctl command turns off the charge pump (isolates the charge pump from the loop filter). This command sets the CHPMPTRI bit (Bit 6) in the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx). It should be used only in the event of loss of clock to provide enough time for shutting down the chip.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx and MC56F801x.

Design/Implementation: The *OCCS_TURN_OFF_CHARGE_PUMP* ioctl command is implemented as a macro.

Example 5-9. OCCS_TURN_OFF_CHARGE_PUMP

```
ioctl(OCCS, OCCS_TURN_OFF_CHARGE_PUMP, NULL);
```

This command sets the CHPMPTRI bit (Bit 6) in the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).

5.1.3.10 OCCS_SET_ZCLOCK_SOURCE - set ZCLOCK source

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_ZCLOCK_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-14. OCCS_SET_ZCLOCK SOURCE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter determining the source. Use one of these predefined constants: **on MC56F83xx:** OCCS_POSTSCALER_OUTPUT / OCCS_PRESCALER_OUTPUT **or on MC56F801x and MC56F802x/3x:** OCCS_POSTSCALER_OUTPUT / OCCS_MSTR_OSC_OUTPUT **or on MC56F800x:** OCCS_PLL_OSC_OUTPUT / OCCS_MSTR_OSC_OUTPUT

Description: The *OCCS_SET_ZCLOCK_SOURCE* ioctl command sets the ZCLOCK source, which determines the clock source to the DSC core. The ZCLOCK source can be either a prescaler output, when OCCS_PRESCALER_OUTPUT is used as a parameter or a postscaler output, when OCCS_POSTSCALER_OUTPUT is used as a parameter on MC56F83xx.

Or it can be directly a master oscillator clock, when OCCS_MSTR_OSC_OUTPUT is used as a parameter or a postscaler output, when OCCS_POSTSCALER_OUTPUT is used as a parameter on MC56F801x and MC56F802x/3x.

On MC56F800x can by directly a master oscillator clock, when OCCS_MSTR_OSC_OUTPUT is used as a parameter or a PLL output clock, when OCCS_PLL_OSC_OUTPUT is used as a parameter.

This command writes to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx), Bits 1-0 (ZSRC).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_SET_ZCLOCK_SOURCE* ioctl command is implemented as a macro.

Example 5-10. OCCS_SET_ZCLOCK_SOURCE

```
ioctl(OCCS, OCCS_SET_ZCLOCK_SOURCE, OCCS_PRESCALER_OUTPUT);
```

This code selects the prescaler output as DSC core clock.

5.1.3.11 OCCS_GET_ZCLOCK_SOURCE - get ZCLOCK source

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_GET_ZCLOCK_SOURCE,
              NULL);
```

Arguments:

Table 5-15. OCCS_GET_ZCLOCK SOURCE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_GET_ZCLOCK_SOURCE* ioctl command returns the state of ZSRC bits in the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx). These bits indicate the current ZCLOCK Source.

Returns: The return value is the content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx), where all bits, except the ZSRC, are cleared:

0x0001 - current ZCLOCK Source is a Prescaler output (**on MC56F83xx**) or MSTR_OSC (**on MC56F80xx**)

0x0002 - current ZCLOCK Source is a Postscaler output (**on MC56F83xx, MC56F801x and MC56F802x/3x**)

PLL output (**on MC56F800x**)

0x0000 or 0x0003 - Synchronizing in progress

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_GET_ZCLOCK_SOURCE* ioctl command is implemented as a macro.

Example 5-11. OCCS_GET_ZCLOCK_SOURCE

```
UWord16 temp;

temp = ioctl(OCCS, OCCS_GET_ZCLOCK_SOURCE, NULL);
```

This code stores the state of the ZSRC bits in variable temp.

5.1.3.12 OCCS_READ_FLAG - read the status of selected flags

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_READ_FLAG, UWord16 param);
```

Arguments:

Table 5-16. OCCS_READ_FLAG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the flag. Use consolidation of these predefined constants: OCCS_STATUS_LOCK_LOST_INT1 | OCCS_STATUS_LOCK_LOST_INT0 | OCCS_STATUS_CLOCK_LOST | OCCS_STATUS_LOCK_1 | OCCS_STATUS_LOCK_0 OCCS_STATUS_POWER_DOWN OCCS_STATUS_ZCLOCK **and on MC56F800x additional:** OCCS_STATUS_CRYSTAL_READY

Description: The *OCCS_READ_FLAG* ioctl command returns the status of selected flags (bits) from the PLL Status register (PLLSR on MC56F83xx, STAT on MC56F80xx). The permissible flags are: PLL Loss of Lock Interrupt 1 (LOLI1) - Bit 15 (parameter OCCS_STATUS_LOCK_LOST_INT1 used), PLL Loss of Lock Interrupt 0 (LOLI0) - Bit 14 (parameter OCCS_STATUS_LOCK_LOST_INT0 used), Loss of Clock (LOCI) - Bit 13 (parameter OCCS_STATUS_CLOCK_LOST used), Loss of Lock 1 (LCK1) - Bit 6 (parameter OCCS_STATUS_LOCK_1 used), Loss of Lock 0 (LCK0) - Bit 5 (parameter OCCS_STATUS_LOCK_0 used), PLL Power Down (PLLPD) - Bit 4 (parameter OCCS_STATUS_POWER_DOWN used) and Clock Source (ZSRC) - Bit 1, 0 (parameter OCCS_STATUS_ZCLOCK used). On MC56F800x can by read Oscillator Ready (COSC_RDY) - Bit 2 (parameter OCCS_STATUS_CRYSTAL_READY used).

Returns: The state of the selected flags. The returned value is the content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx), where all bits, except the ones specified as parameter param, are cleared:

0x0000 - no flag is set

0x8000 | 0x4000 | 0x2000 | 0x0040 | 0x0020 | 0x0010 | 0x0004 | 0x0003 - if LOLI1 | LOLI0 | LOCI | LCK1 | LCK0 | PLLPDN | COSC_RDY | ZCRCS is set (where | is an operator meaning logical OR).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_READ_FLAG* ioctl command is implemented as a macro.

Example 5-12. OCCS_READ_FLAG

```
while (!ioctl(OCCS, OCCS_READ_FLAG, OCCS_STATUS_LOCK_1))
      ;
```

This code waits until Loss of Lock 1 (LCK1) flag in the PLL Status Register (PLLSR) is set (until PLL is locked).

5.1.3.13 OCCS_CLEAR_FLAG - clear selected flags

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_CLEAR_FLAG,
           UWord16 param);
```

Arguments:

Table 5-17. OCCS_CLEAR_FLAG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the flag. Use the consolidation of these pre-defined constants: OCCS_STATUS_LOCK_LOST_INT1 | OCCS_STATUS_LOCK_LOST_INT0 | OCCS_STATUS_CLOCK_LOST

Description: The *OCCS_CLEAR_FLAG* ioctl command clears the selected flags (bits) from the PLL Status register (PLLSR on MC56F83xx, STAT on MC56F80xx). The allowed flags are: PLL Loss of Lock Interrupt 1 (LOLI1) - Bit 15 (parameter OCCS_STATUS_LOCK_LOST_INT1 used), PLL Loss of Lock Interrupt 0 (LOLI0) - Bit 14 (parameter OCCS_STATUS_LOCK_LOST_INT0 used) and Loss of Clock (LOCI) - Bit 13 (parameter OCCS_STATUS_CLOCK_LOST used). The specified flags are cleared by writing a one at a flag position in the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_CLEAR_FLAG* ioctl command is implemented as a macro.

Example 5-13. OCCS_CLEAR_FLAG

```
ioctl(OCCS, OCCS_CLEAR_FLAG, OCCS_STATUS_LOCK_LOST_INT1);
```

This code clears bit 15 (LOLI1) in the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx) by writing one to this bit.

5.1.3.14 OCCS_GET_IPBUS_FREQ - get IPBus Clock frequency

Call(s):

```
UWord32 ioctl(const int *pModuleBase, OCCS_GET_IPBUS_FREQ,
          UInt32 param);
```

Arguments:

Table 5-18. OCCS_GET_IPBUS_FREQ ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Crystal oscillator frequency [Hz].

Description: The *OCCS_GET_IPBUS_FREQ* ioctl command returns the current IPBus Clock frequency. The returned frequency is calculated according to the current settings in the OCCS registers. See Section 5.1.2.1, “OCCS frequency calculation.” for reference on expressions which are used to calculate the IPBus Clock frequency.

Returns: IPBus Clock frequency [Hz].

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_GET_IPBUS_FREQ* ioctl command is implemented as a function call. Note that for the *EXTCLK* external crystal frequency, the core clock frequency is calculated directly by the C preprocessor and is available as *OCCS_CORE_CLOCK* macro.

Example 5-14. OCCS_GET_IPBUS_FREQ

```
UWord32 IPBusFreq;

IPBusFreq = ioctl(OCCS, OCCS_GET_IPBUS_FREQ, 8000000L);
```

This code stores the current IPBus Clock frequency in variable IPBusFreq.

5.1.3.15 OCCS_SET_LORTP - set Loss of Reference Timer Period (LORTP)

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_LORTP,
           UWord16 param);
```

Arguments:

Table 5-19. OCCS_SET_LORTP ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Loss of Reference Timer Period (LORTP). Permissible range is 0 to 15.

Description: The *OCCS_SET_LORTP* ioctl command sets the Loss of Reference Timer Period (LORTP). The command writes passed parameter param to the PLL Divide-by Register (PLLDB), Bits 15-12 (LORTP).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_SET_LORTP* ioctl command is implemented as a macro.

Example 5-15. OCCS_SET_LORTP

```
ioctl(OCCS, OCCS_SET_LORTP, 4);
```

This code sets the Loss of Reference Timer Period (LORTP) to value of 4.

5.1.3.16 OCCS_WRITE_CONTROL_REG - write to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx)

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WRITE_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-20. OCCS_WRITE_CONTROL_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Value to write to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).

Description: The *OCCS_WRITE_CONTROL_REG* ioctl command writes a 16-bit value (parameter param) to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_WRITE_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-16. OCCS_WRITE_CONTROL_REG

```
ioctl(OCCS, OCCS_WRITE_CONTROL_REG, 0x0882);
```

This code writes 0x0882 to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx).

5.1.3.17 OCCS_WRITE_DIVIDE_BY_REG - write to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx)

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WRITE_DIVIDE_BY_REG,
           UWord16 param);
```

Arguments:

Table 5-21. OCCS_WRITE_DIVIDE_BY_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Value to write to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).

Description: The *OCCS_WRITE_DIVIDE_BY_REG* ioctl command writes a 16-bit value (parameter param) to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_WRITE_DIVIDE_BY_REG* ioctl command is implemented as a macro.

Example 5-17. OCCS_WRITE_DIVIDE_BY_REG

```
ioctl(OCCS, OCCS_WRITE_DIVIDE_BY_REG, 0x0C0F);
```

This code writes 0x0C0F to the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).

5.1.3.18 OCCS_WRITE_OSC_CONTROL_REG - write to the Oscillator Control Register (OSCTL)

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WRITE_OSC_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-22. OCCS_WRITE_OSC_CONTROL_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Value to write to the Oscillator Control Register (OSCTL).

Description: The OCCS_WRITE_OSC_CONTROL_REG ioctl command writes a 16-bit value (parameter param) to the Oscillator Control Register (OSCTL).

Returns: None.

Range Issues: None.

Special Issues: The content of the OSCTL register differs with or without Relaxation Oscillator included on the chip.

Design/Implementation: The *OCCS_WRITE_OSC_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-18. OCCS_WRITE_OSC_CONTROL_REG

```
ioctl(OCCS, OCCS_WRITE_OSC_CONTROL_REG, 0x0201);
```

This code writes 0x0201 to the Oscillator Control Register (OSCTL) on the chip with the Relaxation Oscillator.

5.1.3.19 OCCS_READ_CONTROL_REG - read the PLL Control Register (PLLSR on MC56F83xx, STAT on MC56F80xx)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_READ_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-23. OCCS_READ_CONTROL_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_READ_CONTROL_REG* ioctl command returns the content of the PLL Control Register (PLLCR).

Returns: Content of the PLL Control Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-19. OCCS_READ_CONTROL_REG

```
UWord16 uRegVal;

uRegVal = ioctl(OCCS, OCCS_READ_CONTROL_REG, NULL);
```

This code stores the content of the PLL Control Register (PLLSR on MC56F83xx, STAT on MC56F80xx) in variable uRegVal.

5.1.3.20 OCCS_READ_DIVIDE_BY_REG - read the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_READ_DIVIDE_BY_REG,
          NULL);
```

Arguments:

Table 5-24. OCCS_READ_DIVIDE_BY_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_READ_DIVIDE_BY_REG* ioctl command returns the content of the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).

Returns: Content of the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_READ_DIVIDE_BY_REG* ioctl command is implemented as a macro.

Example 5-20. OCCS_READ_DIVIDE_BY_REG

```
UWord16 uRegVal;

uRegVal = ioctl(OCCS, OCCS_READ_DIVIDE_BY_REG, NULL);
```

This code stores the content of the PLL Divide-by Register (PLLDB on MC56F83xx, DIVBY on MC56F80xx) in variable uRegVal.

5.1.3.21 OCCS_READ_STATUS_REG - read the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_READ_STATUS_REG,
          NULL);
```

Arguments:

Table 5-25. OCCS_READ_STATUS_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_READ_STATUS_REG* ioctl command returns the content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

Returns: Content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *OCCS_READ_STATUS_REG* ioctl command is implemented as a macro.

Example 5-21. OCCS_READ_STATUS_REG

```
UWord16 uRegVal;

uRegVal = ioctl(OCCS, OCCS_READ_STATUS_REG, NULL);
```

This code stores the content of the PLL Status Register (PLLSR on MC56F83xx, STAT on MC56F80xx) in variable uRegVal.

5.1.3.22 OCCS_READ_OSC_CONTROL_REG - read the Oscillator Control Register (OSCTL)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_READ_OSC_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-26. OCCS_READ_OSC_CONTROL_REG ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The OCCS_READ_OSC_CONTROL_REG ioctl command returns the content of the Oscillator Control Register (OSCTL).

Returns: Content of the Oscillator Control Register (OSCTL).

Range Issues: None.

Special Issues: The content of the OSCTL register differs with or without Relaxation Oscillator included on the chip.

Design/Implementation: The *OCCS_READ_OSC_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-22. OCCS_READ_OSC_CONTROL_REG

```
UWord16 uRegOscCtrl;

uRegOscCtrl = ioctl(OCCS, OCCS_READ_OSC_CONTROL_REG, NULL);
```

This code stores the content of the Oscillator Control Register (OSCTL) in variable uRegOscCtrl.

5.1.3.23 OCCS_POWER_MODE - control the power usage of the crystal oscillator

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_POWER_MODE,
           UWord16 param);
```

Arguments:

Table 5-27. OCCS_POWER_MODE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the desired mode. Use one of these predefined constants: OCCS_HIGH_POWER / OCCS_LOW_POWER

Description: The *OCCS_POWER_MODE* ioctl command controls the power usage of the crystal oscillator. This command sets the COHL bit (Bit 13) in the Oscillator Control Register (OSCTL) when OCCS_LOW_POWER is used as a parameter (low power mode is the desired mode when a crystal is used). This command clears the above mentioned bit in case of OCCS_HIGH_POWER (high power mode is required when a resonator is used).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_POWER_MODE* ioctl command is implemented as a macro.

Example 5-23. OCCS_POWER_MODE

```
ioctl(OCCS, OCCS_POWER_MODE, OCCS_HIGH_POWER);
```

This code sets low power mode when a crystal is used.

5.1.3.24 OCCS_SHUTDOWN - shutdown all system clocks

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SHUTDOWN,
           UWord16 param);
```

Arguments:

Table 5-28. OCCS_SHUTDOWN ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Value to write to the Shutdown Register (SHUTDOWN).

Description: The OCCS_SHUTDOWN ioctl command writes a 16-bit value (parameter param) to the Shutdown Register (SHUTDOWN) followed by an idle loop. See the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx and MC56F801x.

Design/Implementation: The *OCCS_SHUTDOWN* ioctl command is implemented as a macro.

Example 5-24. OCCS_SHUTDOWN

```
ioctl(OCCS, OCCS_SHUTDOWN, 0xDEAD);
```

This code writes 0xDEAD to the Shutdown Register (SHUTDOWN).

5.1.3.25 OCCS_SET_PRESCALER_CLOCK - set the prescaler clock source

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_PRESCALER_CLOCK,
           UWord16 param);
```

Arguments:

Table 5-29. OCCS_SET_ZCLOCK OCCS_SET_PRESCALER_CLOCK ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter determining the source. Use one of these predefined constants: OCCS_INTERNAL_RELAX_OSC / OCCS_CRYSTAL_OSC

Description: The *OCCS_SET_PRESCALER_CLOCK* ioctl command sets the prescaler clock source. It can be selected to be either the Internal Relaxation Oscillator, when OCCS_INTERNAL_RELAX_OSC is used as a parameter or Crystal Oscillator, when OCCS_CRYSTAL_OSC is used as a parameter. This command writes to the PLL Control Register (PLLCR on MC56F83xx, CTRL on MC56F80xx), Bit 2 (PRECS).

Returns: None.

Range Issues: None.

Special Issues: This command is valid only when the Relaxation Oscillator is included on the chip.

Design/Implementation: The *OCCS_SET_PRESCALER_CLOCK* ioctl command is implemented as a macro.

Example 5-25. OCCS_SET_PRESCALER_CLOCK

```
ioctl(OCCS, OCCS_SET_PRESCALER_CLOCK, OCCS_INTERNAL_RELAX_OSC);
```

This code selects the Relaxation Oscillator as an input clock to the prescaler.

5.1.3.26 OCCS_INTERNAL_RELAX_OSC_OPERATION - enable or power-down the Relaxation Oscillator

Call(s):

```
void ioctl(const int *pModuleBase,
     OCCS_INTERNAL_RELAX_OSC_OPERATION, UWord16 param);
```

Arguments:

Table 5-30. OCCS_INTERNAL_RELAX_OSC_OPERATION ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Parameter to select the desired action. Use one of these predefined constants: OCCS_ENABLE / OCCS_DISABLE

Description: The *OCCS_INTERNAL_RELAX_OSC_OPERATION* ioctl command enables or powers-down the internal Relaxation Oscillator. This command sets the ROPD bit (Bit 15) in the Oscillator Control Register (OSCTL) when OCCS_DISABLE is used as a parameter. This command clears the above mentioned bit in case of OCCS_ENABLE.

Returns: None.

Range Issues: None.

Special Issues: This command is valid only when the Relaxation Oscillator is included on the chip.

Design/Implementation: The *OCCS_INTERNAL_RELAX_OSC_OPERATION* ioctl command is implemented as a macro.

Example 5-26. OCCS_INTERNAL_RELAX_OSC_OPERATION

```
ioctl(OCCS, OCCS_INTERNAL_RELAX_OSC_OPERATION, OCCS_ENABLE);
```

This code enables the internal Relaxation Oscillator.

5.1.3.27 OCCS_ADJUST_RELAX_OSC_FREQ - adjust the Relaxation Oscillator frequency

Call(s):

```
void ioctl(const int *pModuleBase,
     OCCS_ADJUST_RELAX_OSC_FREQ, UWord16 param);
```

Arguments:

Table 5-31. OCCS_ADJUST_RELAX_OSC_FREQ ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Value to adjust the frequency.

Description: The *OCCS_ADJUST_RELAX_OSC_FREQ* ioctl command adjusts the internal Relaxation Oscillator frequency by changing the size of the internal capacitor. This command modifies the TRIM bits (Bits 9-0) in the Oscillator Control Register (OSCTL). Reset sets these bits to $200, centering the range of possible adjustment. Incrementing these bits by one decreases the clock period time by 0.078 percent of the unadjusted value. Decrementing this register by one increases the period time by 0.078 percent.

Returns: None.

Range Issues: UWord16 value must be in range of 0 to 1023 (0x000 - 0x3FF).

Special Issues: This command is valid only when the Relaxation Oscillator is included on the chip.

Design/Implementation: The *OCCS_ADJUST_RELAX_OSC_FREQ* ioctl command is implemented as a macro.

Example 5-27. OCCS_ADJUST_RELAX_OSC_FREQ

```
ioctl(OCCS, OCCS_ADJUST_RELAX_OSC_FREQ, 0x20F);
```

This code adjusts the internal Relaxation Oscillator frequency by applying 0x20F as parameter value.

5.1.3.28 OCCS_TRIM_RELAX_OSC_8MHZ - adjust the Relaxation Oscillator frequency to 8MHz using the factory settings

Call(s):

```
void ioctl(const int *pModuleBase,OCCS_TRIM_RELAX_OSC_8MHZ,
           NULL);
```

Arguments:

Table 5-32. OCCS_TRIM_RELAX_OSC_8MHZ ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_TRIM_RELAX_OSC_8MHZ* ioctl command adjusts the internal Relaxation Oscillator frequency to 8Mhz by applying the factory settings. This command reads the factory TRIM value from the internal flash and modifies the TRIM bits (Bits 9-0) in the Oscillator Control Register (OSCTL).

The standard startup code of the application created with DSP56800E_Quick_Start is capable of setting the trimming value automatically, when this is enabled in the Graphical Configuration Tool and the *appconfig.h* file. Also, the startup is capable of re-calculating the trimming value to achieve alternate oscillator frequency, approximately in the range of 6..13 MHz.

Returns: None.

Range Issues: None.

Special Issues: This command is valid only when the Relaxation Oscillator is included on the chip.

Design/Implementation: The *OCCS_TRIM_RELAX_OSC_8MHZ* ioctl command is implemented as a macro.

Example 5-28. OCCS_TRIM_RELAX_OSC_8MHZ

```
ioctl(OCCS, OCCS_TRIM_RELAX_OSC_8MHZ, NULL);
```

This code adjusts the internal Relaxation Oscillator frequency to 8MHz.

5.1.3.29 OCCS_DIRECT_CLOCK_MODE - enable or disable direct clock input on XTAL

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_DIRECT_CLOCK_MODE,
           UWord16 param);
```

Arguments:

Table 5-33. OCCS_DIRECT_CLOCK_MODE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants: **OCCS_ENABLE** ... when clock source is connected on the XTAL pin **OCCS_DISABLE** ... when crystal or resonator is connected on the EXTAL and XTAL pins

Description: The *OCCS_DIRECT_CLOCK_MODE* ioctl command enables or disables the direct clock input on the XTAL pin. Note that the OCCS_SELECT_EXT_CLOCK_SOURCE ioctl command needs to be used first to switch clock input to OCCS_CLKIN_OSC.

Then the *OCCS_DIRECT_CLOCK_MODE* command may be used to enable or disable direct clock input on XTAL. Use command parameter according to the Table 5-33 above.

This command writes directly to the CLKMODE bit of the OCCS Oscillator Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command has no effect when clock input is not in the OCCS_CLKIN_OSC mode. See OCCS_SELECT_EXT_CLOCK_SOURCE ioctl command for more details. This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_DIRECT_CLOCK_MODE* ioctl command is implemented as a macro.

Example 5-29. OCCS_DIRECT_CLOCK_MODE

```
ioctl(OCCS, OCCS_SELECT_EXT_CLOCK_SOURCE, OCCS_CLKIN_OSC);
ioctl(OCCS, OCCS_DIRECT_CLOCK_MODE, OCCS_ENABLE);
```

This code selects direct clock source on the XTAL pin. Please note that the appropriate GPIO pin needs to be configured properly as well.

5.1.3.30 OCCS_SELECT_EXT_CLOCK_SOURCE - select clock source

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SELECT_EXT_CLOCK_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-34. OCCS_SELECT_EXT_CLOCK_SOURCE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **on MC56F802x/3x:** **OCCS_CLKIN_PRI** ... primary clock source on GPIO_B6 **OCCS_CLKIN_ALT** ... alternate clock source on GPIO_B5 **OCCS_CLKIN_OSC** ... crystal/resonator clock on the EXTAL/XTAL pins or direct clock source on the XTAL pin **on MC56F800x:** **OCCS_CLKIN_CLKIN** ... clock source on GPIO_B6 **OCCS_CLKIN_EXTAL** ... crystal/resonator clock on the EXTAL/XTAL pins or direct clock source on the XTAL pin

Description: The *OCCS_SELECT_EXT_CLOCK_SOURCE* ioctl command selects clock source which will be used as external clock source. On the MC56F802x/3x devices, there are three clock sources to be chosen from. Two direct clock inputs (primary and alternate) and one crystal/resonator input. The latter can be further configured as a standard EXTAL/XTAL input or as a direct clock input on the XTAL pin. See the OCCS_DIRECT_CLOCK_MODE ioctl command for more details. On the MC56F800x devices, there are only two clock sources to be chosen from. One is direct clock input and second is crystal/resonator input. The second input can be configured as a standard EXTAL/XTAL input or as a direct clock input on the XTAL pin.

The external clock source selected by OCCS_SELECT_EXT_CLOCK_SOURCE may then become an official MSTR_OSC clock by using the OCCS_SET_PRESCALER_CLOCK command.

This command writes directly to the EXT_SEL bit-field of the OCCS Oscillator Control Register.

Returns: None.

Range Issues: None.

Special Issues: Selected clock source pins need to be further configured in appropriate GPIO modules. This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_SELECT_EXT_CLOCK_SOURCE* ioctl command is implemented as a macro.

Example 5-30. OCCS_SELECT_EXT_CLOCK_SOURCE

```
ioctl(OCCS, OCCS_SELECT_EXT_CLOCK_SOURCE, OCCS_CLKIN_PRI);
```

This code selects direct clock source on primary CLKIN pin (typically GPIO_B6 on MC56F802x/3x). Please note that the appropriate GPIO pin needs to be configured properly as well.

5.1.3.31 OCCS_WPROTECT_PLL_SETTINGS - write-protect PLL settings

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WPROTECT_PLL_SETTINGS,
           UWord16 param);
```

Arguments:

Table 5-35. OCCS_WPROTECT_PLL_SETTINGS ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **OCCS_ENABLE** ... enable protection (may be changed later) **OCCS_ENABLE_PERMANENT** ... enable protection until reset **OCCS_DISABLE** ... disable protection (may be re-enabled later) **OCCS_DISABLE_PERMANENT** ... disable protection until reset

Description: The *OCCS_WPROTECT_PLL_SETTINGS* ioctl command write-protects the PLL-related configuration bits. Depending on the parameter value, the protection may be activated or deactivated permanently (until next reset).

The OCCS_WPROTECT_PLL_SETTINGS command protects the PLLPDN, LOCIE and LORTP bits in the OCCS Control and in Divide-By registers.

This command writes directly to the PLLEP bit-field of the OCCS Protection Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_WPROTECT_PLL_SETTINGS* ioctl command is implemented as a macro.

Example 5-31. OCCS_WPROTECT_PLL_SETTINGS

```
ioctl(OCCS, OCCS_WPROTECT_PLL_SETTINGS, OCCS_ENABLE_PERMANENT);
```

This code write-protects the PLL configuration bits. It will not be possible to disable this protection until the next reset occurs.

5.1.3.32 OCCS_WPROTECT_OSC_SETTINGS - write-protect oscillator settings

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WPROTECT_OSC_SETTINGS,
           UWord16 param);
```

Arguments:

Table 5-36. OCCS_WPROTECT_OSC_SETTINGS ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **OCCS_ENABLE** ... enable protection (may be changed later) **OCCS_ENABLE_PERMANENT** ... enable protection until reset **OCCS_DISABLE** ... disable protection (may be re-enabled later) **OCCS_DISABLE_PERMANENT** ... disable protection until reset

Description: The *OCCS_WPROTECT_OSC_SETTINGS* ioctl command write-protects the Oscillator-related configuration registers and bits. Depending on the parameter value, the protection may be activated or deactivated permanently (until next reset).

The OCCS_WPROTECT_OSC_SETTINGS command protects the OCCS Oscillator Control Register and the PRESC bit in the OCCS Control Register.

This command writes directly to the OSCEP bit-field of the OCCS Protection Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_WPROTECT_OSC_SETTINGS* ioctl command is implemented as a macro.

Example 5-32. OCCS_WPROTECT_OSC_SETTINGS

```
ioctl(OCCS, OCCS_WPROTECT_OSC_SETTINGS, OCCS_DISABLE_PERMANENT);
```

This code disables write-protection of the Oscillator configuration bits. It will not be possible to enable this protection until next the reset occurs.

5.1.3.33 OCCS_WPROTECT_CLK_SETTINGS - write-protect clock frequency settings

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_WPROTECT_CLK_SETTINGS,
           UWord16 param);
```

Arguments:

Table 5-37. OCCS_WPROTECT_CLK_SETTINGS ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **OCCS_ENABLE** ... enable protection (may be changed later) **OCCS_ENABLE_PERMANENT** ... enable protection until reset **OCCS_DISABLE** ... disable protection (may be re-enabled later) **OCCS_DISABLE_PERMANENT** ... disable protection until reset

Description: The *OCCS_WPROTECT_CLK_SETTINGS* ioctl command write-protects the Clock frequency-related configuration bits. Depending on the parameter value, the protection may be activated or deactivated permanently (until next reset).

The OCCS_WPROTECT_CLK_SETTINGS command protects the PLLCOD and ZSRCS bits in the OCCS Control and in Divide-By registers.

This command writes directly to the FREQEP bit-field of the OCCS Protection Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_WPROTECT_CLK_SETTINGS* ioctl command is implemented as a macro.

Example 5-33. OCCS_WPROTECT_CLK_SETTINGS

```
ioctl(OCCS, OCCS_WPROTECT_CLK_SETTINGS, OCCS_ENABLE);
```

This code write-protects the Frequency configuration bits. It will be possible to disable this protection later.

5.1.3.34 OCCS_SET_CLOCK_CHECK - Clock checking function

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SET_CLOCK_CHECK,
           UWord16 param);
```

Arguments:

Table 5-38. OCCS_SET_CLOCK_CHECK ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **OCCS_ENABLE** ... enable clock checking function **OCCS_DISABLE** ... disable clock checking function

Description: The *OCCS_SET_CLOCK_CHECK* ioctl command starts and stops clock checking function. This command sets the CHK_ENA bit (Bit 15) in the External clock check reference register (CLKCHKR). This command enables clock checking function and resets counters REF_COUNT and TARGET_CNT, when parameter is OCCS_ENABLE. When clock checking function finished, bit CHK_ENA is cleared and in counters REF_COUNT and TARGET_CNT are valid values. Parameter OCCS_DISABLE stops clock checking function.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_SET_CLOCK_CHECK* ioctl command is implemented as a macro.

Example 5-34. OCCS_SET_CLOCK_CHECK

```
ioctl(OCCS, OCCS_SET_CLOCK_CHECK, OCCS_ENABLE);
```

This code enables clock checking function and resets counters REF_COUNT and TARGET_CNT.

5.1.3.35 OCCS_TEST_CLOCK_CHECK - test if clock checking function finished

Call(s):

```
UWord16 ioctl(const int *pModuleBase, OCCS_TEST_CLOCK_CHECK,
          NULL);
```

Arguments:

Table 5-39. OCCS_TEST_CLOCK_CHECK ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_TEST_CLOCK_CHECK* ioctl command tests, if clock checking function finished. This command tests the CHK_ENA bit (Bit 15) in the External clock check reference register (CLKCHKR). If value of the CHK_ENA is 1, the clock checking function is working. When value of the bit is changed to 0, clock checking function is finished and in the and in counters REF_COUNT and TARGET_CNT are valid values.

Returns: State of CHK_ENA bit .

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_TEST_CLOCK_CHECK* ioctl command is implemented as a macro.

Example 5-35. OCCS_TEST_CLOCK_CHECK

```
while (ioctl(OCCS, OCCS_TEST_CLOCK_CHECK, NULL))
;
```

This code waits until clock checking function is finished - bit CHK_ENA is cleared.

5.1.3.36 OCCS_READ_CLOCK_CHECK_REFERENCE - Read clock checking function reference counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
OCCS_READ_CLOCK_CHECK_REFERENCE,NULL);
```

Arguments:

Table 5-40. OCCS_READ_CLOCK_CHECK_REFERENCE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_READ_CLOCK_CHECK_REFERENCE* ioctl command reads result of clock checking function for internal reference clock. This command reads the REFERENCE_CNT bits (Bit 14-0) in the External clock check reference register (CLKCHKR).

Returns: Value of REFERENCE_CNT in CLKCHKR register.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_READ_CLOCK_CHECK_REFERENCE* ioctl command is implemented as a macro.

Example 5-36. OCCS_READ_CLOCK_CHECK_REFERENCE

```
UWord16 reference = ioctl(OCCS, OCCS_READ_CLOCK_CHECK_REFERENCE,
NULL);
```

This code returns result of clock checking function for internal reference clock.

5.1.3.37 OCCS_READ_CLOCK_CHECK_TARGET - Read clock checking function target counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
OCCS_READ_CLOCK_CHECK_TARGET,NULL);
```

Arguments:

Table 5-41. OCCS_READ_CLOCK_CHECK_TARGET ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.

Description: The *OCCS_READ_CLOCK_CHECK_TARGET* ioctl command reads result of clock checking function for external clock. This command reads the TARGET_CNT bits (Bit 7-0) in the External clock check target register (CLKCHKT).

Returns: Value of TARGET_CNT in CLKCHKT register.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *OCCS_READ_CLOCK_CHECK_TARGET* ioctl command is implemented as a macro.

Example 5-37. OCCS_READ_CLOCK_CHECK_TARGET

```
UWord16 target = ioctl(OCCS, OCCS_READ_CLOCK_CHECK_TARGET, NULL);
```

This code returns result of clock checking function for external clock.

5.1.3.38 OCCS_SELECT_FREQ_RANGE - Select frequency range of the crystal oscillator

Call(s):

```
void ioctl(const int *pModuleBase, OCCS_SELECT_FREQ_RANGE,UWord16
param);
```

Arguments:

Table 5-42. OCCS_SELECT_FREQ_RANGE ioctl call arguments

pModuleBase	in	The OCCS module identifier. Use OCCS.
param	in	Use one of the following constants **OCCS_32KHZ_CRYSTAL** ... Select 32kHz mode **OCCS_1MHZ_TO_16MHZ_CRYSTAL** ... Select 1MHz to 16MHz mode

Description: The *OCCS_SELECT_FREQ_RANGE* ioctl command sets frequency range of the crystall oscillator. This command sets the RANGE bit (Bit 11) in the Oscillator control register (OSCTL). This command can set the crystal oscillator range from 1 MHz to 16Mhz, when OCCS_1MHZ_TO_16MHZ_CRYSTAL is used as a parameter or set range for 32kHz crystal, when OCCS_32KHZ_CRYSTAL is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *OCCS_SELECT_FREQ_RANGE* ioctl command is implemented as a macro.

Example 5-38. OCCS_SELECT_FREQ_RANGE

```
ioctl(OCCS, OCCS_SELECT_FREQ_RANGE, OCCS_1MHZ_TO_16MHZ);
```

This code sets the crystal oscillator range from 1MHZ to 16MHz.

5.1.4 OCCS Driver Sample Application

The OCCS driver application is designed for Freescale/Motorola EVMs and is intended to illustrate the usage of this driver by a real example.

The application shows a static and a dynamic initialization of the OCCS module. The static initialization is performed by the OCCS_INIT command. See configurable items in *appconfig.h*, which are written into the OCCS registers. The dynamic initialization is shown using the OCCS_SET_CORE_CLOCK command.

The OCCS module offers the possibility to generate an interrupt in the event of loss of clock. This feature is also demonstrated in this sample application. The loss of clock is simulated by pressing the IRQA button. After the IRQA button is pressed, a loss of clock interrupt is generated. Inside the interrupt service routine the CHPMPTRI bit in the PLL Control Register (PLLCR) is set. Afterwards the user should shut down the chip. Setting the CHPMPTRI bit isolates the charge pump from the loop filter, allowing the PLL output to drift slowly away. The frequency of the blinking LED is slowly drifting the same way as the PLL output what can be seen on the debugging LEDs. The debugging LEDs are driven by the PWM module, which is configured according to the *pwm_demo* application.

The OCCS driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\occs_demo* directory and consists of the application project *occs_demo.mcp* and the source code for the application *main.c*.

Example 5-39. OCCS driver sample application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Thu, 08/Feb/2007, 16:07:33
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 8 MHz
    VCO frequency: 256 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Enable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0882
#define OCCS_PLLDB_INIT                  0xFC9F
#define OCCS_REQUIRED_LOCK_MODE          0x40

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
```

```
                          TMR A: Enable , TMR B: Enable , TMR C: Enable
                          TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                          CAN: Enable , ADC A: Enable , ADC B: Enable
                          EMI: Enable
.*/
#define SIM_GPS_INIT                    0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                0x0005
#define SEMI_CSBAR1_INIT                0x0005
#define SEMI_CSBAR2_INIT                0x0005
#define SEMI_CSOR0_INIT                 0x1FC3
#define SEMI_CSOR1_INIT                 0x1BA3
#define SEMI_CSOR2_INIT                 0x1DA3

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Falling-edge sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0001
#define INT_VECTOR_ADDR_17              occs_isr
#define INT_PRIORITY_LEVEL_17           INTC_LEVEL0
#define INT_VECTOR_ADDR_21              occs_isr
#define INT_PRIORITY_LEVEL_21           INTC_LEVEL0
#define INT_VECTOR_ADDR_78              pwm_reload_isr
#define INT_PRIORITY_LEVEL_78           INTC_LEVEL0

/*.
    PWM_A Configuration
-----------------------------------------
    PWM module operation: Enabled
    Prescaler: /8 , PWM clock period: 1 us
    PWM frequency: 15.25925 Hz , Period: 65.534 ms
```

```
    PWM Deadtime: 46.875 us
    PWM Reload Frequency: Every opportunity
    Alignment: Center
    Debug Mode Operation: Stop
    Wait Mode Operation: Stop
    Deadtime Correction Method: Manual correction (no correction)
    Half Cycle Reload: Disabled
    Keep Hardware Acceleration Register Bits Writable: Disabled
    Output Pad Enable: Enabled
    Load OK: Yes
    Swap & Mask Mode: DSP56F80X compatible
    Write Protection: No
    Top Polarity   : Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Bottom Polarity: Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Chann. coupling: Channels 4-5: Complementary
                     Channels 2-3: Complementary
                     Channels 0-1: Complementary
    Clearing Mode:   Fault 0 pin: Manual
                     Fault 1 pin: Manual
                     Fault 2 pin: Manual
    Asymmetric Operation: Channels 0-1: Off (Correction Method used only)
                          Channels 2-3: Off (Correction Method used only)
                          Channels 4-5: Off (Correction Method used only)
.*/
#define PWM_A_PMCTL_INIT                0x00E3
#define PWM_A_PMOUT_INIT                0x8000
#define PWM_A_PWMCM_INIT                0x7FFF
#define PWM_A_PMDEADTM_INIT             0x002F
#define PWM_A_PMDISMAP1_INIT            0x0000
#define PWM_A_PMDISMAP2_INIT            0x0000
#define PWM_A_USE_PWMVAL                1
#define PWM_A_PWMVAL0_INIT              0x4000
#define PWM_A_PWMVAL2_INIT              0x4000
#define PWM_A_PWMVAL4_INIT              0x4000

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: ISA0 , PullUp: Enable ,
    Pin  9: Function: ISA1 , PullUp: Enable ,
    Pin 10: Function: ISA2 , PullUp: Enable ,
.*/
#define GPIO_C_DDR_INIT                 0x000F
#define GPIO_C_PER_INIT                 0x07F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
```

```
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  7: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_DDR_INIT                 0x00C0
#define GPIO_D_PER_INIT                 0x1F01


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-40. OCCS driver sample application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application demonstrating usage of OCCS driver.
*
*   The application is configured to run from the external clock source (crystal).
*   In the main loop, the application periodically switches the core clock
*   between 5MHz and 7.5MHz. The PWM is also configured so you should be able
*   to observe how the clock frequency changes.
*
*   The loss-of-reference-clock interrupt is enabled, so when the crystal
*   is detached (remove the EXTAL or XTAL jumper), the PLL charge pump is
*   turned off and PLL is let to slowly drift while still providing a
*   usable clock (which gives a time to shotdown the application). The RED
*   LED is set to signal this situation.
*
*   The loss-of-reference-clock interrupt is also connected to the IRQ_A
*   interrupt button - so you don't need to physically detach the crystal.
*   This button will turn PLL charge pump off also if device is running from
*   an internal relaxation oscillator.
*
* TARGET: MC56F83xx devices
*
*****************************************************************************/

#include "qs.h"

#include "occs.h"
#include "sys.h"
#include "intc.h"
#include "gpio.h"
#include "cop.h"
#include "pwm.h"

/* board-specific LEDs */
#include "../board.h"

/* forward prototypes */
void device_init(void);

/* counter of PWM reload interrupts */
UWord16 pwmReloadsCnt = 0;

#pragma interrupt on

/*
 *  OCCS Interrupt Service Routine - called in the event of loss of clock
 */

void occs_isr(void)
{
    /* NOTE: on devices with an internal relaxation oscillator (e.g. 56F8323), it is
       a good idea to switch to this internal clock source. This should be more
       reliable than the using an unconnected PLL. See OCCS_SET_PRESCALER_CLOCK for
       more details. */
```

```
    /* turn off charge pump, let PLL to slowly drift */
    ioctl(OCCS, OCCS_TURN_OFF_CHARGE_PUMP, NULL);

    /* RED LED */
    ioctl(GPIO_LEDS, GPIO_SET_PIN, LED_R);

    /* clear loss of clock interrupt flag (LOCI) in PLL Status Register (PLLSR) */
    ioctl(OCCS, OCCS_CLEAR_FLAG, OCCS_STATUS_CLOCK_LOST);

    /* shutdown the system */
    archDisableInt();
    while(1) {}
}

/*
 *  PWM reload interrupt, just to provide timing for the main loop
 */

void pwm_reload_isr(void)
{
    ioctl(PWM_A, PWM_CLEAR_RELOAD_FLAG, NULL);

    pwmReloadsCnt ++;
}

#pragma interrupt off


/*
 *  The main
 */

int main(void)
{
    UInt32 ipBusFreq;

    /* initialize SYS, COP and pins */
    device_init();

    /* make sure the RED LED is GPIO output and off */
    ioctl(GPIO_LEDS, GPIO_SETAS_GPIO, LED_R);
    ioctl(GPIO_LEDS, GPIO_SETAS_OUTPUT, LED_R);
    ioctl(GPIO_LEDS, GPIO_CLEAR_PIN, LED_R);

    /* initialize PWM */
    ioctl(PWM_A, PWM_INIT, NULL);

   /* get the IPBus Clock frequency as set in startup code (according to appconfig.h)
*/
    ipBusFreq = ioctl(OCCS, OCCS_GET_IPBUS_FREQ, EXTCLK);

    /* configure Interrupt Controller (IPR) */
    ioctl(INTC, INTC_INIT, NULL);

    /* enable loss-of-clock interrupt */
    ioctl(OCCS, OCCS_CLEAR_FLAG, OCCS_STATUS_CLOCK_LOST);
    ioctl(OCCS, OCCS_INT_ENABLE, OCCS_LOSS_OF_CLOCK_INT);

    /* enable maskable interrupts in Status Register (SR), bits I1 and I0 */
    archEnableInt();

    while(1)
    {
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
```

```
        /* change clock every 32 PWM periods */
        if(!(pwmReloadsCnt & 0x1f))
        {
            /* green LED toggle */
            ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_G);

           /* switch between two frequencies, this should be visible on the PWM LEDs
*/
            if(pwmReloadsCnt & 0x20)
            {
                /* set 7.5MHz operation (240MHz VCO) */
                ioctl(OCCS, OCCS_SET_CORE_CLOCK, OCCS_CLOCK_IN_DIVIDE_BY_1 |
                    OCCS_CLOCK_OUT_DIVIDE_BY_8 | 30);
            }
            else
            {
                /* set 5MHz operation (160MHz VCO) */
                ioctl(OCCS, OCCS_SET_CORE_CLOCK, OCCS_CLOCK_IN_DIVIDE_BY_2 |
                    OCCS_CLOCK_OUT_DIVIDE_BY_8 | 40);
            }

            /* recalculate the immediate clock frequency */
            ipBusFreq = ioctl(OCCS, OCCS_GET_IPBUS_FREQ, EXTCLK);

            /* skip this PWM reload */
            pwmReloadsCnt++;
        }
    }
}

/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.2 INTC Driver

This section describes the API for the MC56F83xx and MC56F80xx Interrupt Controller (INTC) on-chip module. The functionality of the INTC module itself is described in the Data Sheets of each particular device.

5.2.1 Introduction

The MC56F83xx/MC56F80xx interrupt system consists of the Interrupt unit in the processor core and of the Interrupt Controller Module (INTC). The processor core supports four interrupt priority levels with hardware support for interrupt nesting. Priority levels 0-2 are maskable using the dedicated bits in the core Status Register. Priority level 3 is non maskable. Depending on a processor device, the INTC peripheral module enables up to 85 interrupt sources to be mapped to selected priority level and provides the necessary interface to the processor core.

See MC56F83xx Core Reference Manual or Section 2.5 on page 2-24 of this document for more information about interrupts and interrupt processing.

5.2.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.2.3.

Table 5-43. INTC Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F83xx
INTC (INTC_BASE)	0xF060	0xF0E0	0xF1A0

5.2.2.1 API Definition

The following header files are needed in order to use the INTC device driver:

Required Header File(s):

```
#include "qs.h"
#include "intc.h"
```

The following information may be found in the header file *intc.h*.

Public Data Structure(s):

none

5.2.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static INTC module configuration by the driver routines. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-56 for more details.

Table 5-44. INTC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
INT_VECTOR_ADDR_n ('n' for any interrupt except the ones with fixed priority levels)	function identifier	Installs the specified function as the interrupt service routine for the interrupt source n.
INT_PRIORITY_LEVEL_n ('n' for any interrupt except the ones with fixed priority levels)	one of: INTC_DISABLED INTC_LEVEL0 INTC_LEVEL1 INTC_LEVEL2 INTC_LEVEL3	Prepares the n-th entry in the initialization value for the appropriate Interrupt Priority Register. When this macro is not defined for given n, the INTC_DISABLED is taken as default.
INTC_ICTL_INIT	UWord16	Initial value for the INTC Control Register.
INTC_FIM0_INIT	UWord16 interrupt number	initial values for the INTC Fast Interrupt Match Registers for fast interrupts 0 and 1.
INTC_FIM1_INIT		
INTC_FIVA0_INIT	function identifier	Initial values for the Fast Interrupt Vector Address Registers. These values installs the fast service routines for fast interrupts 0 and 1. When not defined, the value of INT_VECTOR_ADDR_n where n=INTC_FIMx_INIT is used.
INTC_FIVA1_INIT		

5.2.2.3 API Specification

This section specifies the usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
```

Description: The *ioctl* call sets the INTC registers.

Arguments:

Table 5-45. INTC Driver Arguments - ioctl

pModuleBase	in	INTC module identifier. Use INTC.
cmd	in	Commands found in *itcn.h* which are used to modify registers of INTC. See Table 5-46.
pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-46. ioctl commands

Cmd	pParam	Return	Description
INTC_INIT	NULL	None	Applies the appconfig.h static configuration to the respective INTC registers.
INTC_INTERRUPTS	INTC_ENABLE / INTC_DISABLE	None	Globally enables or disables the interrupts processed by the INTC module. Note that this is not equal as enabling interrupts in the processor core.

Table 5-46. ioctl commands (Continued)

Cmd	pParam	Return	Description
INTC_SET_IPL_n ('n' for any interrupt except the ones with fixed priority levels)	one of: INTC_DISABLED / INTC_LEVEL0 / INTC_LEVEL1 / INTC_LEVEL2 / INTC_LEVEL3	None	Sets the interrupt priority level for interrupt n. Handles the difference between Level 0,1,2 and Level 1,2,3 interrupts.
INTC_SET_IPL_n_RAW ('n' for any interrupt except the ones with fixed priority levels)	0...3	None	Sets the two bit IPL value for a given interrupt. Note that such a value has different meaning for Level 0,1,2 and Level 1,2,3 interrupts.
INTC_GET_IPL_n_RAW ('n' for any interrupt except the ones with fixed priority levels)	NULL	UWord16 (0...3)	Gets the two bit IPL value of given interrupt.
INTC_SET_FASTINT0	UWord16 interrupt number	None	Sets the interrupt specified in the parameter as the fast interrupt 0 or 1.
INTC_SET_FASTINT1			
INTC_SET_FASTINT0_VEC	function identifier	None	Registers the given function as the interrupt service routine for the fast interrupt 0 or 1.
INTC_SET_FASTINT1_VEC			
INTC_GET_PENDING_FLAG	UWord16 interrupt number	None	Tests whether given interrupt processing is currently pending.
INTC_READ_CONTROL_REG	NULL	UWord16	Reads the value of the INTC Control Register ICTL.
INTC_GET_INT_STATE	NULL	UWord16	Returns the state of the interrupt being sent to the core.
INTC_GET_INT_LEVEL	NULL	UWord16	Returns the priority level of the interrupt being sent to the core.
INTC_GET_INT_NUMBER	NULL	UWord16	Returns the number of the highest priority pending interrupt.
INTC_READ_IRQPINS	NULL	INTC_IRQA | INTC_IRQB	Returns the immediate state of the IRQA and IRQB processor pins.
INTC_SELECT_EDGE_MODE	INTC_IRQA | INTC_IRQB	None	Selects interrupts for which to set falling edge sensitive mode.
INTC_SELECT_LEVEL_MODE	INTC_IRQA | INTC_IRQB	None	Selects interrupts for which to set low level sensitive mode.

5.2.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.2.3.1 INTC_INIT - initialize interrupt controller

Call(s):

```
void ioctl(const int *pModuleBase, INTC_INIT, NULL);
```

Arguments:

Table 5-47. INTC_INIT ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.

Description: The *INTC_INIT* ioctl command calls the INTC initialization routine in which the *appconfig.h* static configuration values are applied to the respective registers. See Table 5-44 for the list of all configuration items.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_INIT* ioctl command is implemented as a function call.

Example 5-41. INTC_INIT

```
ioctl(INTC, INTC_INIT, NULL);
```

This code initializes the INTC module by the values defined in *appconfig.h*. The *appconfig.h* file can be edited manually or generated by the Graphical Configuration Tool.

5.2.3.2 INTC_INTERRUPTS - enable or disable interrupt processing

Call(s):

```
void ioctl(const int *pModuleBase, INTC_INTERRUPTS, bool param);
```

Arguments:

Table 5-48. INTC_INTERRUPTS ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Use INTC_ENABLE to enable or INTC_DISABLE to disable the interrupt processing.

Description: The *INTC_INTERRUPTS* ioctl command enables or disables the interrupt processing by the INTC module. This command directly writes the INT_DIS bit of the INTC Control Register (ICTL).

Returns: None.

Range Issues: None.

Special Issues: Note that there are additional interrupt enable bits on the 56F8xxx core. You may access these bits with *archEnableInt* or *archDisableInt* macros.

Design/Implementation: The *INTC_INTERRUPTS* ioctl command is implemented as a macro.

Example 5-42. INTC_INTERRUPTS

```
ioctl(INTC, INTC_INTERRUPTS, INTC_DISABLE);
```

This code disables the interrupt processing.

5.2.3.3 INTC_SET_IPL_n - sets the interrupt priority level

Call(s):

```
void ioctl(const int *pModuleBase, INTC_SET_IPL_n, param);

// ... where n is the interrupt number
```

Arguments:

Table 5-49. INTC_SET_IPL_n ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Use one of the special constants (macros) to specify the desired interrupt priority level for the interrupt n. INTC_DISABLED / INTC_LEVEL0 / INTC_LEVEL1 / INTC_LEVEL2 / INTC_LEVEL3

Description: The *INTC_SET_IPL_n* ioctl command writes the IPL bits in the Interrupt Priority Register for the interrupt number *n*. This has the effect of setting the interrupt priority level for given interrupt.

Returns: None.

Range Issues: None.

Special Issues: As there is a two-bit priority level value for each interrupt and there are five levels to be set (Disabled, Level 0, 1, 2 and 3), there is always one interrupt level inaccessible for each interrupt. The *INTC_SET_IPL_n* implementation contains run-time validation of requested priority level. In case an invalid level is being assigned by the *ioctl* call, the *INTC_InvalidIPL* function is invoked and the program execution stops at the debug halt instruction (the IPL is not affected).

Because of the runtime validation, the *INTC_SET_IPL_n* command generates an optimal code for constant parameters only. Use *INTC_SET_IPL_n_RAW* command if you wish to use a variable as the command parameter.

Design/Implementation: The *INTC_SET_IPL_n* ioctl command is implemented as a macro.

Example 5-43. INTC_SET_IPL_n

```
ioctl(INTC, INTC_SET_IPL_34, INTC_LEVEL1);
```

This code sets the interrupt priority level 1 for the interrupt number 34.

5.2.3.4 INTC_SET_IPL_n_RAW - sets the interrupt priority level bits

Call(s):

```
void ioctl(const int *pModuleBase, INTC_SET_IPL_n_RAW,
     UWord16 param);

// ... where n is the interrupt number
```

Arguments:

Table 5-50. INTC_SET_IPL_n_RAW ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	The numeric value. Two lowest significant bits are directly written into the IPL bit-field for given interrupt

Description: The *INTC_SET_IPL_n_RAW* ioctl command writes directly the *param* value into the IPL bits in the Interrupt Priority Register for the interrupt number *n*. For most interrupts this means assigning one of priority levels: Disabled, Level 0, 1 and 2. However, for some system interrupts the Level 0 is not permissible and the two bit value assigns priority levels: Disabled, Level 1, 2 and 3. Refer to the device data sheet for more details.

There is no run-time validations of the param value. This command can be used to restore interrupt priority level saved to a variable using the *INTC_GET_IPL_n_RAW* command.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_SET_IPL_n_RAW* ioctl command is implemented as a macro.

Example 5-44. INTC_SET_IPL_n_RAW

```
UWord16 ipl = ioctl(INTC, INTC_GET_IPL_34_RAW, NULL);
...
ioctl(INTC, INTC_SET_IPL_34_RAW, ipl);
```

This code saves and later restores the interrupt priority level for interrupt number 34.

5.2.3.5 INTC_GET_IPL_n_RAW - gets the interrupt priority level bits

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_GET_IPL_n_RAW, NULL);

// ... where n is the interrupt number
```

Arguments:

Table 5-51. INTC_GET_IPL_n_RAW ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.

Description: The *INTC_GET_IPL_n_RAW* ioctl command returns a two-bit IPL value for given interrupt. For the most of interrupts, the two-bit value represents one of the interrupt levels: Disabled, Level 0, 1 and 2. For selected system interrupts the value represents the interrupt levels Disabled, Level 1, 2 and 3. Refer to the device data sheet for more details.

Use this command in conjunction with *INTC_SET_IPL_n_RAW* command to save and restore the interrupt priority level in run-time.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_GET_IPL_n_RAW* ioctl command is implemented as a macro.

Example 5-45. INTC_GET_IPL_n_RAW

```
UWord16 ipl = ioctl(INTC, INTC_GET_IPL_34_RAW, NULL);
...
ioctl(INTC, INTC_SET_IPL_34_RAW, ipl);
```

This code saves and later restores the interrupt priority level for interrupt number 34.

5.2.3.6 INTC_SET_FASTINTx - installs the fast interrupt 0 or 1

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_SET_FASTINT0,
      UWord16 param);
UWord16 ioctl(const int *pModuleBase, INTC_SET_FASTINT1,
      UWord16 param);
```

Arguments:

Table 5-52. INTC_SET_FASTINTx ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	The number of interrupt which is to be set as the fast interrupt.

Description: The *INTC_SET_FASTINT0* and *INTC_SET_FASTINT1* ioctl commands register the specified interrupts as the fast interrupts 0 or 1. These commands write directly to the INTC Fast Interrupt Match Registers (FIM0 and FIM1).

Returns: None.

Range Issues: None.

Special Issues: There is always one interrupt level inaccessible for any given interrupt. The *INTC_SET_FASTINTx* implementation contains compile-time validation of requested priority level.

Design/Implementation: The *INTC_SET_FASTINTx* ioctl commands are implemented as macros.

Example 5-46. INTC_SET_FASTINT0

```
ioctl(INTC, INTC_SET_FASTINT0, 34);
```

This code sets interrupt #34 as the fast interrupt 0.

5.2.3.7 INTC_SET_FASTINTx_VEC - installs the fast interrupt service routine

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_SET_FASTINT0_VEC,
      void (*param)());
UWord16 ioctl(const int *pModuleBase, INTC_SET_FASTINT1_VEC,
      void (*param)());
```

Arguments:

Table 5-53. INTC_SET_FASTINTx_VEC ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	An identifier (name) of the function which is to be installed as fast interrupt service routine.

Description: The *INTC_SET_FASTINT0_VEC* and *INTC_SET_FASTINT1_VEC* ioctl commands register the specified function as the fast interrupt service routines for fast interrupts 0 or 1. These commands write directly to the INTC Fast Interrupt Vector Address Registers (FIVA0 and FIVA1).

Returns: None.

Range Issues: None.

Special Issues: Keep in mind that there are several limitations in the way how the fast interrupt service routines can be coded and that the *frtid* instruction must be used instead of *rti* or *rtid*.

Design/Implementation: The *INTC_SET_FASTINTx_VEC* ioctl commands are implemented as macros.

Example 5-47. INTC_SET_FASTINT0_VEC

```
ioctl(INTC, INTC_SET_FASTINT0_VEC, MyFastISR);
```

This code sets the function named MyFastISR as the service routine for the fast interrupt 0. The correct prototype of the function is

```
void MyFastISR(void);
```

5.2.3.8 INTC_GET_PENDING_FLAG - tests whether interrupt is pending

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_GET_PENDING_FLAG,
          UWord16 param);
```

Arguments:

Table 5-54. INTC_GET_PENDING_FLAG ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	The index of an interrupt for which to obtain an interrupt pending flag.

Description: The *INTC_GET_PENDING_FLAG* ioctl command reads and tests the bit in the INTC Pending Register for requested interrupt.

Returns: The UWord16 value containing the interrupt pending bit from the appropriate INTC Pending Register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_GET_PENDING_FLAG* ioctl command is implemented as a macro.

Example 5-48. INTC_GET_PENDING_FLAG

```
if(ioctl(INTC, INTC_GET_PENDING_FLAG, 34))
{
      // interrupt 34 is pending - it is probably masked
      // or disabled, otherwise the ISR would be called
}
```

This code tests whether the interrupt #34 is pending.

5.2.3.9 INTC_READ_CONTROL_REG - read the INTC Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_READ_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-55. INTC_READ_CONTROL_REG ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Not used.

Description: The *INTC_READ_CONTROL_REG* ioctl command returns the immediate value of INTC Control Register (ICTL).

Returns: ICTL Register value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-49. INTC_READ_CONTROL_REG

```
UWord16 ictl = ioctl(INTC, INTC_READ_CONTROL_REG, NULL);
```

This code reads the ITCL register.

5.2.3.10 INTC_GET_INT_STATE - get current interrupt state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_GET_INT_STATE, NULL);
```

Arguments:

Table 5-56. INTC_GET_INT_STATE ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Not used.

Description: The *INTC_GET_INT_STATE* ioctl command reads and tests the INT bit of the INTC Control Register (ICTL). This bit is set when an interrupt request is currently being sent to the processor core.

Returns: True (non-zero) when interrupt request is being sent to the processor code. Zero when no interrupts are waiting for processing.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_GET_INT_STATE* ioctl command is implemented as a macro.

Example 5-50. INTC_GET_INT_STATE

```
if(ioctl(INTC, INTC_GET_INT_STATE, NULL))
{
      // there is an interrupt to be processed
}
```

This code tests the ICTL.INT bit.

5.2.3.11 INTC_GET_INT_LEVEL - get current interrupt priority level

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_GET_INT_LEVEL, NULL);
```

Arguments:

Table 5-57. INTC_GET_INT_LEVEL ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Not used.

Description: The *INTC_GET_INT_LEVEL* ioctl command reads the value of IPIC field of the INTC Control Register (ICTL). This value indicates the priority level of interrupt which is currently being sent to the processor core.

Returns: Interrupt priority level as the number 0...3.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_GET_INT_LEVEL* ioctl command is implemented as a macro.

Example 5-51. INTC_GET_INT_LEVEL

```
if(ioctl(INTC, INTC_GET_INT_LEVEL, NULL) > 1)
{
      // there is level2 of level3 interrupt being or
      // waiting to be processed
}
```

This code tests the ICTL.IPIC bit filed.

5.2.3.12 INTC_GET_INT_NUMBER - get current interrupt priority level

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_GET_INT_NUMBER, NULL);
```

Arguments:

Table 5-58. INTC_GET_INT_NUMBER ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Not used.

Description: The *INTC_GET_INT_NUMBER* ioctl command reads the value of VAB field of the INTC Control Register (ICTL). This value indicates the number of interrupt which is currently being sent to the processor core.

Returns: Interrupt number.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_GET_INT_NUMBER* ioctl command is implemented as a macro.

Example 5-52. INTC_GET_INT_NUMBER

```
if(ioctl(INTC, INTC_GET_INT_NUMBER, NULL) == 34)
{
      // interrupt #34 is just being or
      // waiting to be processed
}
```

This code tests the ICTL.IPIC bit filed.

5.2.3.13 INTC_READ_IRQPINS - read immediate state of external interrupt pins

Call(s):

```
UWord16 ioctl(const int *pModuleBase, INTC_READ_IRQPINS, NULL);
```

Arguments:

Table 5-59. INTC_READ_IRQPINS ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Not used

Description: The *INTC_READ_IRQPINS* ioctl command reads the immediate state of the IRQA and IRQB processor pins. The value is taken directly from INTC Control Register (ICTL) IRQA and IRQB STATE bits.

Returns: The state of the pins. Use the INTC_IRQA and INTC_IRQB constants to test the returned value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_READ_IRQPINS* ioctl command is implemented as a macro.

Example 5-53. INTC_READ_IRQPINS

```
if(ioctl(INTC, INTC_READ_IRQPINS, NULL) & INTC_IRQA)
{
      // the IRQA input pin is currently driven high
}
```

This code tests the state of the IRQA processor pin.

5.2.3.14 INTC_SELECT_EDGE_MODE - select falling-edge mode of the external interrupts

Call(s):

```
void ioctl(const int *pModuleBase, INTC_SELECT_EDGE_MODE,
           UWord16 param);
```

Arguments:

Table 5-60. INTC_SELECT_EDGE_MODE ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Parameter specifying the interrupts for which to set falling-edge sensitive mode. This can be a combination of the following constants: INTC_IRQA | INTC_IRQB

Description: The *INTC_SELECT_EDGE_MODE* ioctl command sets the falling-edge trigger mode of the external interrupts IRQA and/or IRQB. This command writes directly to the INTC Control Register IRQA and IRQB EDG bits.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_SELECT_EDGE_MODE* ioctl command is implemented as a macro.

Example 5-54. INTC_SELECT_EDGE_MODE

```
ioctl(INTC, INTC_SELECT_EDGE_MODE, INTC_IRQA);
```

This code sets the falling-edge sensitivity mode for the external interrupt IRQA.

5.2.3.15 INTC_SELECT_LEVEL_MODE - select low-level mode of the external interrupts

Call(s):

```
void ioctl(const int *pModuleBase, INTC_SELECT_LEVEL_MODE,
           UWord16 param);
```

Arguments:

Table 5-61. INTC_SELECT_LEVEL_MODE ioctl call arguments

pModuleBase	in	The INTC module identifier. Use INTC.
param	in	Parameter specifying the interrupts for which to set low-level sensitive mode. This can be a combination of the following constants: INTC_IRQA | INTC_IRQB

Description: The *INTC_SELECT_LEVEL_MODE* ioctl command sets the low-level trigger mode of the external interrupts IRQA and/or IRQB. This command writes directly to the INTC Control Register IRQA and IRQB EDG bits.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *INTC_SELECT_LEVEL_MODE* ioctl command is implemented as a macro.

Example 5-55. INTC_SELECT_LEVEL_MODE

```
ioctl(INTC, INTC_SELECT_LEVEL_MODE, INTC_IRQB);
```

This code sets the low-level sensitivity mode for the external interrupt IRQB.

5.2.4 INTC Driver Application

The INTC driver application is designed for Freescale/Motorola EVM's and is intended to illustrate the usage of the external interrupts $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ by a real example. Examples of how to use the other features of an INTC driver (i.e. setting the interrupt priority, enabling/disabling interrupt channels) can be found in sample applications of the other drivers, because the interrupt priority and the interrupt channels are always related to a concrete peripheral.

The INTC driver application can be found in e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\irq_demo* directory and consists of the application project *irq_demo.mcp* and the source code for the application *main.c*.

Example 5-56. INTC driver application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Mon, 26/Sep/2005, 11:28:11
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02010004L

/*.
    OCCS Configuration
----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
```

```
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output Mode: Off: Tristated
.*/
#define SIM_GPS_INIT                        0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                    0x0005
#define SEMI_CSBAR1_INIT                    0x0005
#define SEMI_CSBAR2_INIT                    0x0005
#define SEMI_CSOR0_INIT                     0x1FC3
#define SEMI_CSOR1_INIT                     0x1BA3
#define SEMI_CSOR2_INIT                     0x1DA3

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                      0x0000
#define INT_VECTOR_ADDR_17                  irqA_isr
#define INT_PRIORITY_LEVEL_17               INTC_LEVEL1

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
```

```
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                  0x1FC1


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-57. INTC driver application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application demonstrating the use of external interrupt
*                            IRQA. Use IRQA button to toggle RED LED. GREEN LED is
flashing.
*
* TARGET: MC56F8346 device
*
*****************************************************************************/

#include "qs.h"

#include "occs.h"
#include "intc.h"
#include "gpio.h"

/* few EVM specific defines */

#define LED_RED BIT_0
#define LED_GREENBIT_2
#define LEDS (LED_RED | LED_GREEN)

/*****************************************************************************
            IRQA Interrupt service routine
*****************************************************************************/

#pragma interrupt
void irqA_isr(void)
{
            /* toggle RED on port C */
            ioctl(GPIO_C, GPIO_TOGGLE_PIN, LED_RED);
}


/*****************************************************************************
            main
*****************************************************************************/
void main(void)
{
            int i;

            /* Setup LEDs GPIO on port C (could be done also via appconfig.h) */
            ioctl(GPIO_C, GPIO_SETAS_GPIO, LEDS);
            ioctl(GPIO_C, GPIO_SETAS_OUTPUT, LEDS);
            ioctl(GPIO_C, GPIO_WRITE_DATA, 0);

            /* configure Interrupt Controller (IPR) */
            ioctl(INTC, INTC_INIT, NULL);

            /* configure IRQ mode */
            ioctl(INTC, INTC_SELECT_EDGE_MODE, INTC_IRQA );

            /* enable maskable interrupts in Status Register (SR), bits I1 and I0 */
            archEnableInt();
```

```
        while (1)
        {
               /* keep GREEN flashing */
               ioctl(GPIO_C, GPIO_TOGGLE_PIN, LED_GREEN);

               for(i=0; i<100; i++)
                      archDelay(0xffff);
        }
}
```

5.3 WINTC Driver

This section describes the API for the MC56F800x Interrupt Controller (WINTC) on-chip module. The functionality of the WINTC module itself is described in the Data Sheets of each particular device.

5.3.1 Introduction

The MC56F800x interrupt system consists of the Interrupt unit in the processor core and of the Interrupt Controller Module (WINTC). The processor core supports four interrupt priority levels with hardware support for interrupt nesting. Priority levels 0-2 are maskable using the dedicated bits in the core Status Register. Priority level 3 is non maskable. Depending on a processor device, the WINTC peripheral module enables up to 85 interrupt sources. Peripheral interrupts are mapped to priority level 0. Up to three level 0 peripheral interrupts can be re-assigned as level 1 interrupts: USER1, USER2 and USER3. When an interrupt is re-assigned, its original vector becomes inactive, and the ISR address must be replaced in USER1/2/3 instead. In similar fasion, up to three level 0 peripheral interrupts can be re-assigned as level 2 interrupts: USER4, USER5 and USER6. If activated, USER6 can act as a fast interrupt. See the **56F800x Peripheral Reference Manual** for more details.

5.3.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.3.3.

Table 5-62. WINTC Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
WINTC (WINTC_BASE)	0xF040	N/A	N/A	N/A

5.3.2.1 API Definition

The following header files are needed in order to use the WINTC device driver:

Required Header File(s):

```
#include "qs.h"
#include "wintc.h"
```

The following information may be found in the header file w*intc.h*.

Public Data Structure(s):

none

5.3.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static WINTC module configuration by the driver routines. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-69 for more details.

Table 5-63. WINTC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
INT_VECTOR_ADDR_n ('n' for any interrupt except the ones with fixed priority levels)	function identifier	Installs the specified function as the interrupt service routine for the interrupt source n.
WINTC_ICSR_INIT	UWord16	Initial value for the WINTC Control Register.
WINTC_VBA_INIT	UWord16	Initial value for the Vector Base Address Register.
WINTC_IAR0_INIT	UWord16	Initial value for the Interrut Assigment Register 0.
WINTC_IAR1_INIT	UWord16	Initial value for the Interrut Assigment Register 1.
WINTC_IAR2_INIT	UWord16	Initial value for the Interrut Assigment Register 2.

5.3.2.3 API Specification

This section specifies the usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
```

Description: The *ioctl* call sets the WINTC registers.

Arguments:

Table 5-64. WINTC Driver Arguments - ioctl

pModuleBase	in	WINTC module identifier. Use WINTC.
cmd	in	Commands found in *itcn.h* which are used to modify registers of WINTC. See Table 5-65.
pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-65. ioctl commands

Cmd	pParam	Return	Description
WINTC_INIT	NULL	None	Applies the appconfig.h static configuration to the respective WNTC registers.

Table 5-65. ioctl commands (Continued)

Cmd	pParam	Return	Description
WINTC_INTERRUPTS	WINTC_ENABLE / WINTC_DISABLE	None	Globally enables or disables the interrupts processed by the WINTC module. Note that this is not equal as enabling interrupts in the processor core.
WINTC_GET_INT_STATE	NULL	UWord16	Returns the state of the interrupt being sent to the core.
WINTC_GET_INT_LEVEL	NULL	UWord16	Returns the priority level of the interrupt being sent to the core.
WINTC_GET_INT_NUMBER	NULL	UWord16	Returns the number of the highest priority pending interrupt.
WINTC_ASSIGN_USER1_ VECTOR WINTC_ASSIGN_USER2_ VECTOR WINTC_ASSIGN_USER3_ VECTOR WINTC_ASSIGN_USER4_ VECTOR WINTC_ASSIGN_USER5_ VECTOR WINTC_ASSIGN_USER6_ VECTOR	Word16 (10..44)	None	Sets an user interrupt vectors. The user interrupts USER1, USER2 and USER3 re-assigns the peripheral interrupt priority to level 1. The user interrupts USER4, USER5 and USER6 re-assigns the peripheral interrupt priority to level 2. The user interrupt USER6 can by used as fast interrupt.
WINTC_SET_INT_EOnCE_STEP_ COUNTER	WINTC_ENABLE / WINTC_DISABLE	None	Enables/disables step counter interrupt.
WINTC_SET_INT_EOnCE_ RECEIVER_REGISTER_FULL	WINTC_ENABLE / WINTC_DISABLE	None	Enables/disables receiver register full interrupt.
WINTC_SET_INT_EOnCE_ TRANSMIT_REGISTER_EMPTY	WINTC_ENABLE / WINTC_DISABLE	None	Enables/disables transmit register empty interrupt.
WINTC_SET_INT_EOnCE_ TRACE_BUFFER	WINTC_ENABLE / WINTC_DISABLE	None	Enables/disables trace buffer interrupt.
WINTC_SET_INT_EOnCE_ BREAKPOINT_UNIT0	WINTC_ENABLE / WINTC_DISABLE	None	Enables/disables breakpoint unit 0 interrupt.

5.3.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.3.3.1 WINTC_INIT - initialize interrupt controller

Call(s):

```
void ioctl(const int *pModuleBase, WINTC_INIT, NULL);
```

Arguments:

Table 5-66. WINTC_INIT ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.

Description: The W*INTC_INIT* ioctl command calls the WINTC initialization routine in which the *appconfig.h* static configuration values are applied to the respective registers. See Table 5-63 for the list of all configuration items.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The W*INTC_INIT* ioctl command is implemented as a function call.

Example 5-58. WINTC_INIT

```
ioctl(WINTC, WINTC_INIT, NULL);
```

This code initializes the WINTC module by the values defined in *appconfig.h*. The *appconfig.h* file can be edited manually or generated by the Graphical Configuration Tool.

5.3.3.2 WINTC_INTERRUPTS - enable or disable interrupt processing

Call(s):

```
void ioctl(const int *pModuleBase, WINTC_INTERRUPTS, bool param);
```

Arguments:

Table 5-67. WINTC_INTERRUPTS ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the interrupt processing.

Description: The W*INTC_INTERRUPTS* ioctl command enables or disables the interrupt processing by the WINTC module. This command directly writes the INT_DIS bit of the WINTC Control Register (ICSR).

Returns: None.

Range Issues: None.

Special Issues: Note that there are additional interrupt enable bits on the 56F8xxx core. You may access these bits with *archEnableInt* or *archDisableInt* macros.

Design/Implementation: The W*INTC_INTERRUPTS* ioctl command is implemented as a macro.

Example 5-59. WINTC_INTERRUPTS

```
ioctl(WINTC, WINTC_INTERRUPTS, WINTC_DISABLE);
```

This code disables the interrupt processing.

5.3.3.3 WINTC_GET_INT_STATE - get current interrupt state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, WINTC_GET_INT_STATE, NULL);
```

Arguments:

Table 5-68. WINTC_GET_INT_STATE ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Not used.

Description: The W*INTC_GET_INT_STATE* ioctl command reads and tests the INT_DIS bit of the WINTC Control Register (ICSR). This bit is set when an interrupt request is currently being sent to the processor core.

Returns: True (non-zero) when interrupt request is being sent to the processor code. Zero when no interrupts are waiting for processing.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *WINTC_GET_INT_STATE* ioctl command is implemented as a macro.

Example 5-60. WINTC_GET_INT_STATE

```
if(ioctl(WINTC, WINTC_GET_INT_STATE, NULL))
{
      // there is an interrupt to be processed
}
```

This code tests the ICSR.INT_DIS bit.

5.3.3.4 WINTC_GET_INT_LEVEL - get current interrupt priority level

Call(s):

```
UWord16 ioctl(const int *pModuleBase, WINTC_GET_INT_LEVEL, NULL);
```

Arguments:

Table 5-69. WINTC_GET_INT_LEVEL ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Not used.

Description: The W*INTC_GET_INT_LEVEL* ioctl command reads the value of IPIC field of the WINTC Control Register (ICSR). This value indicates the priority level of interrupt which is currently being sent to the processor core.

Returns: Interrupt priority level as the number 0...3.

Range Issues: None.

Special Issues: None.

Design/Implementation: The W*INTC_GET_INT_LEVEL* ioctl command is implemented as a macro.

Example 5-61. WINTC_GET_INT_LEVEL

```
if(ioctl(WINTC, WINTC_GET_INT_LEVEL, NULL) > 1)
{
      // there is level2 of level3 interrupt being or
      // waiting to be processed
}
```

This code tests the ICSR.IPIC bit filed.

5.3.3.5 WINTC_GET_INT_NUMBER - get current interrupt priority level

Call(s):

```
UWord16 ioctl(const int *pModuleBase, WINTC_GET_INT_NUMBER,
NULL);
```

Arguments:

Table 5-70. WINTC_GET_INT_NUMBER ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Not used.

Description: The W*INTC_GET_INT_NUMBER* ioctl command reads the value of VAB field of the WINTC Control Register (ICSR). This value indicates the number of interrupt which is currently being sent to the processor core.

Returns: Interrupt number.

Range Issues: None.

Special Issues: None.

Design/Implementation: The W*INTC_GET_INT_NUMBER* ioctl command is implemented as a macro.

Example 5-62. WINTC_GET_INT_NUMBER

```
if(ioctl(WINTC, WINTC_GET_INT_NUMBER, NULL) == 34)
{
      // interrupt #34 is just being or
      // waiting to be processed
}
```

This code tests the ICSR.VAB bit filed.

5.3.3.6 WINTC_ASSIGN_USERx_VECTOR - assign user interrupt vector

Call(s):

```
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER1_VECTOR,
UWord16 param);
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER2_VECTOR,
UWord16 param);
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER3_VECTOR,
UWord16 param);
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER4_VECTOR,
UWord16 param);
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER5_VECTOR,
UWord16 param);
void ioctl(const int *pModuleBase, WINTC_ASSIGN_USER6_VECTOR,
UWord16 param);
```

Arguments:

Table 5-71. WINTC_ASSIGN_USERx_VECTOR ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Parameter selects interrupt vector.

Description: The W*INTC_ASSIGN_USERx_VECTOR* ioctl command re-assigns interrupt vectors to USER1, USER2 and USER3 with interrupt priority level 1 and USER4, USER5 and USER6 with interrupt priority level 2. When an interrupt vector is re-assigned, its original interrupt vector becomes inactive, and the ISR address must be placed in USER 1/2/3/4/5/6 instead. User interrupt USER6 can by used as fast interrupt. Fast interrupt rutine is in fast_int.c file and on the begin of this function can not be JSR or BSR instruction.

Returns: None.

Range Issues: UWord16 value must be in range of 10 to 44.

Special Issues: None.

Design/Implementation: The W*INTC_ASSIGN_USERx_VECTOR* ioctl command is implemented as a macro.

Example 5-63. WINTC_ASSIGN_USERx_VECTOR

```
ioctl(WINTC, WINTC_ASSIGN_USER1_VECTOR, 11)
```

This code re-assigns the interrupt vector 11 to USER1 interrupt with interrupt priority level 1.

5.3.3.7 WINTC_SET_INT_EOnCE_STEP_COUNTER - Enables / disables step counter interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
WINTC_SET_INT_EOnCE_STEP_COUNTER, UWord16 param);
```

Arguments:

Table 5-72. WINTC_SET_INT_EOnCE_STEP_COUNTER ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the step counter interrupt.

Description: The WINTC_SET_INT_EOnCE_STEP_COUNTER ioctl command enables / disables the step counter interrupt.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The WINTC_SET_INT_EOnCE_STEP_COUNTER ioctl command is implemented as a macro.

Example 5-64. WINTC_SET_INT_EOnCE_STEP_COUNTER

```
ioctl(WINTC, WINTC_SET_INT_EOnCE_STEP_COUNTER, WINTC_ENABLE)
```

This code enables the step counter interrupt.

5.3.3.8 WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL - Enables / disables receiver register full interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL, UWord16 param);
```

Arguments:

Table 5-73. WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the receiver register full interrupt.

Description: The WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL ioctl command enables / disables the receiver register full interrupt.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL ioctl command is implemented as a macro.

Example 5-65. WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL

```
ioctl(WINTC, WINTC_SET_INT_EOnCE_RECEIVER_REGISTER_FULL,
WINTC_ENABLE)
```

This code enables the receiver register full interrupt.

5.3.3.9 WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY - Enables / disables transmit register empty interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY, UWord16 param);
```

Arguments:

Table 5-74. WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the transmit register empty interrupt.

Description: The WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY ioctl command enables / disables the transmit register empty interrupt.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY ioctl command is implemented as a macro.

Example 5-66. WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY

```
ioctl(WINTC, WINTC_SET_INT_EOnCE_TRANSMIT_REGISTER_EMPTY,
WINTC_ENABLE)
```

This code enables the transmit register empty interrupt.

5.3.3.10 WINTC_SET_INT_EOnCE_TRACE_BUFFER - Enables / disables the trace buffer interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
WINTC_SET_INT_EOnCE_TRACE_BUFFER, UWord16 param);
```

Arguments:

Table 5-75. WINTC_SET_INT_EOnCE_TRACE_BUFFER ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the trace buffer interrupt.

Description: The WINTC_SET_INT_EOnCE_TRACE_BUFFER ioctl command enables / disables the trace buffer interrupt.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The WINTC_SET_INT_EOnCE_TRACE_BUFFER ioctl command is implemented as a macro.

Example 5-67. WINTC_SET_INT_EOnCE_TRACE_BUFFER

```
ioctl(WINTC, WINTC_SET_INT_EOnCE_TRACE_BUFFER, WINTC_ENABLE)
```

This code enables the trace buffer interrupt.

5.3.3.11 WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0 - Enables / disables the breakpoint unit 0 interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0, UWord16 param);
```

Arguments:

Table 5-76. WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0 ioctl call arguments

pModuleBase	in	The WINTC module identifier. Use WINTC.
param	in	Use WINTC_ENABLE to enable or WINTC_DISABLE to disable the breakpoint unit 0 interrupt.

Description: The WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0 ioctl command enables / disables the breakpoint unit 0 interrupt.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0 ioctl command is implemented as a macro.

Example 5-68. WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0

```
ioctl(WINTC, WINTC_SET_INT_EOnCE_BREAKPOINT_UNIT0, WINTC_ENABLE)
```

This code enables the breakpoint unit 0 interrupt.

5.3.4 INTC Driver Application

The WINTC driver application is designed for Freescale/Motorola DEMO's and is intended to illustrate the usage of the external interrupts $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ by a real example. Examples of how to use the other features of an WINTC driver (i.e. setting the interrupt priority, enabling/disabling interrupt channels) can be found in sample applications of the other drivers, because the interrupt priority and the interrupt channels are always related to a concrete peripheral.

The WINTC driver application can be found in e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8006DEMO\gpio_demo* directory and consists of the application project *irq_demo.mcp* and the source code for the application *main.c*.

Example 5-69. WINTC driver application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2009 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
* Modules Included:
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Mon, 19/Jan/2009, 17:09:43
*
*************************************************************************.*/

#define MC56F8006
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02040002L

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 32 MHz
    VCO frequency: 192 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38848 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
    Enable Loss of Clock COP: Disable
.*/
#define OCCS_CTRL_INIT                  0x0082U
#define OCCS_DIVBY_INIT                 0x2000U
#define OCCS_USE_FACTORY_TRIM           1
#define COP_COPCTL_INIT                 0x0300U

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    SIM - Clock on GPIO: Enable CLKO_0: No ,
                         Enable CLKO_1: No ,
    SIM - Peripheral Clock Enable: HSCPM 0: No , HSCPM 1: No , HSCPM 2: No , ADC 0: No
                                   ADC 1: No , PGA 0: No , PGA 1: No , I2C: No
                                   SCI: No , SPI: No , PWM: No , COP timer: Yes
                                   PDB: No , PIT: No , Timer Channel 0: No , Timer
```

```
Channel 1: No
    SIM - Modules Enabled in Stop: HSCPM 0: No , HSCPM 1: No , HSCPM 2: No , ADC 0: No
                                   ADC 1: No , PGA 0: No , PGA 1: No , I2C: No
                                   SCI: No , SPI: No , PWM: No , COP timer: No
                                   PDB: No , PIT: No , Timer Channel 0: No , Timer
Channel 1: No
    SIM - HS_PERF Peripheral Clk: SCI: No , PWM: No , Timer: No
    SIM - Internal Periph. Source: PSRC0: Comparator 0 output
                                   PSRC1: Comparator 0 output
                                   PSRC2: Comparator 0 output
                                   FAULT1: FAULT1 input pin (GPIO A4)
                                   FAULT2: FAULT2 input pin (GPIO A5)
                                   FAULT3: FAULT3 input pin (GPIO B5)
                                   Timer T0: Package pin (GPIO B4 or D1)
                                   Timer T1: Package pin (GPIO B0, B5 or D3)
                                   C2_WS: Timer T0 output
                                   C1_WS: Timer T0 output
                                   C0_WS: Timer T0 output
    Protection of IPS and GPSxx  : Registers not protected
    Protection of PCE, SD and PCR: Registers not protected
.*/
#define SIM_PCE_INIT                        0x0010U

/*.
    WINTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
.*/
#define WINTC_ICSR_INIT                     0x0000U
#define INT_VECTOR_ADDR_28                  gpio_isr

/*.
    GPIO_A Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  5: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  6: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  7: Function: RESET_B ,
.*/
#define GPIO_A_DDR_INIT                     0x003FU
#define GPIO_A_PER_INIT                     0xFF80U

/*.
    GPIO_B Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active low ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active low ,
    Pin  2: Function: GPIO , Direction: Input , PullUp: Disable , Interrupt: Enable ,
Int.Polarity: Active high ,
    Pin  3: Function: GPIO , Direction: Input , PullUp: Disable , Interrupt: Enable ,
Int.Polarity: Active high ,
    Pin  4: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
```

```
    Pin  5: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: RXD , PullUp: Enable ,
    Pin  7: Function: TXD , PullUp: Enable ,
.*/
#define GPIO_B_PUR_INIT                  0xFFF3U
#define GPIO_B_PER_INIT                  0xFFC0U
#define GPIO_B_IENR_INIT                 0x000CU
#define GPIO_B_IPOLR_INIT                0x0003U


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-70. WINTC driver application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2009 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application which shows how to simply toggle a GPIO LED
*              pin and also how to handle interrupts from GPIO. Upon each GPIO
*              interrupt (coming from a pushbuttons) the red or yellow LED toggle.
*              Green LED toggles periodically in the main application loop.
*
* TARGET: MC56F8xxx devices
*
*****************************************************************************/

#include "qs.h"

#include "sys.h"
#include "wintc.h"
#include "gpio.h"
#include "cop.h"

/* board-specific LEDs and buttons */

/* three basic LEDs on a single port */
#define GPIO_LEDS GPIO_A
#define LED_R     BIT_0
#define LED_Y     BIT_1
#define LED_G     BIT_2

/* GPIO active-low buttons */
#define GPIO_BTNS GPIO_B
#define BTN_0     BIT_2
#define BTN_1     BIT_3

/* forward prototypes */
void device_init(void);

/*
 *  The main
 */

void main(void)
{
    UWord16 i;

    /* initialize SYS, COP and pins */
    device_init();

    /* pins are already configured from the device_init;
       this is an example of run-time configuration of LEDs outputs */
    ioctl(GPIO_LEDS, GPIO_SETAS_GPIO, LED_R | LED_Y | LED_G);
    ioctl(GPIO_LEDS, GPIO_SETAS_OUTPUT, LED_R | LED_Y | LED_G);

    /* run-time configuration of button inputs */
    ioctl(GPIO_BTNS, GPIO_SETAS_GPIO, BTN_0 | BTN_1);
    ioctl(GPIO_BTNS, GPIO_SETAS_INPUT, BTN_0 | BTN_1);
    ioctl(GPIO_BTNS, GPIO_PULLUP_ENABLE, BTN_0 | BTN_1);
```

```
    ioctl(GPIO_BTNS, GPIO_INT_ENABLE, BTN_0 | BTN_1);

    /* configure Interrupt Controller */
    ioctl(WINTC, WINTC_INIT, NULL);

    /* enable interrupts in SR */
    archEnableInt();

    while(1)
    {
        /* wait a while */
        for(i=0; i<100; i++)
            archDelay(0xffff);

        /* toggle green indicator D1 */
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_G);

        /* service COP */
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
    }
}


/*
 *  GPIO interrupt service routine - toggles LEDs
 */

#pragma interrupt on

void gpio_isr(void)
{
    UWord16 irqs = ioctl(GPIO_BTNS, GPIO_READ_INT_PENDING_REG, NULL);

    /* toggle LEDs */
    if(irqs & BTN_0)
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_R);
    if(irqs & BTN_1)
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_Y);

    /* clear interrupt flags */
    ioctl(GPIO_BTNS, GPIO_CLEAR_INT_PENDING, irqs);
}

#pragma interrupt off

/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
```

```
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.4 COP Driver

This section describes the API for the MC56F83xx and MC56F80xx Computer Operating Properly (COP) on-chip module. The functionality of the COP module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.4.1 Introduction

The MC56F83xx/MC56F80xx devices have one COP module (Computer Operating Properly), also referred as watchdog timer. The COP module helps to recover from runaway code. The COP is a free-running counter which, once enabled, is intended to generate a reset on overflow. The software must service the COP periodically in order to clear the counter and prevent a reset. This section describes the COP driver software, providing the low level software layer interfacing hardware with software.

5.4.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.4.3.

Table 5-77. COP Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
COP (COP_BASE)	0xF140	0xF0E0	0xF120	0xF2C0

5.4.2.1 API Definition

The following header files are needed in order to use the COP device driver:

Required Header File(s):

```
#include "qs.h"
#include "cop.h"
```

The following information may be found in the header file *cop.h*.

Public Data Structure(s):

None

5.4.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static COP module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-84 for more details.

Table 5-78. COP Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
COP_COPTO_INIT	UWord16	The initial value of the COP Time-out Register
COP_COPCTL_INIT	UWord16	Initial value of the COP Control Register

5.4.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void Cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void Cmd, void* pParam);
```

Description: The *ioctl* call "changes" the COP device modes or accesses the COP register(s). The third ioctl parameter is either a value or a pointer, depending on the type of Cmd.

Arguments:

Table 5-79. COP Driver Arguments - ioctl

pModuleBase	in	COP module identifier. Use COP.
Cmd	in	Commands found in *cop.h* which are used to modify the COP module status and control registers. See Table 5-80.

Table 5-79. COP Driver Arguments - ioctl

param	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-80. ioctl commands

Cmd	param	Return	Description
COP_INIT	NULL	None	Initializes COP module by data from configuration file (appconfig.h).
COP_DEVICE	COP_ENABLE/ COP_DISABLE	None	Enable/disable COP device.
COP_SET_TIMEOUT	UWord16 Timeout divisor	None	Sets timeout period by writing to the COP Time-out Register.
COP_READ_COUNTER	NULL	UWord16	Reads the content of the Count Register (COPCTR).
COP_CLEAR_COUNTER	NULL	None	Clears COP counter by writing service sequence (0x5555,0xAAAA) to the COP Service Register.
COP_CLEAR_COUNTER_PART1	NULL	None	Writes the first part of the COP counter clearing service sequence (0x5555) to the COP Service Register.
COP_CLEAR_COUNTER_PART2	NULL	None	Writes the second part of the COP counter clearing service sequence (0xAAAA) to the COP Service Register.
COP_RUN_IN_STOP	COP_ENABLE/ COP_DISABLE	None	Configures COP device to run in STOP mode (COP_ENABLE) or to stop in STOP mode (COP_DISABLE).
COP_RUN_IN_WAIT	COP_ENABLE/ COP_DISABLE	None	Configures COP device to run in WAIT mode (COP_ENABLE) or to stop in WAIT mode (COP_DISABLE).
COP_WRITE_PROTECT	NULL	None	When this command is issued all successive COP commands have no effect until RESET (except the COP_CLEAR_COUNTER).

Table 5-80. ioctl commands (Continued)

Cmd	param	Return	Description
MC56F802x/3x and MC56F800x: COP_SET_CLOCK_SOURCE	**on MC56F802x/3x:** COP_MSTR_OSC / COP_RLX_OSC / COP_IPBUS_CLK **on MC56F800x:** COP_1KHZ / COP_IPBUS/ COP_COSC/ COP_ROSC	None	Sets COP clock source.
MC56F802x/3x and MC56F800x: COP_LOR_WATCHDOG	COP_ENABLE / COP_DISABLE	None	Enables or disables loss-of-reference clock watchdog.
MC56F800x only: COP_SET_CLOCK_PRESCALER	COP_DIV1/ COP_DIV16/ COP_DIV256/ COP_DIV1024	None	Sets COP clock prescaler

5.4.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.4.3.1 COP_INIT - static configuration of the COP

Call(s):

```
void ioctl(const int *pModuleBase, COP_INIT, NULL);
```

Arguments:

Table 5-81. COP_INIT ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Parameter not used.

Description: The *COP_INIT* ioctl command executes the initialization function, which initializes all configured COP registers by the values defined in *appconfig.h*. It is intended for static configuration of the COP module after power-up. This command should be issued immediately after power-up (RESET).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_INIT* ioctl command is implemented as a function call. Only necessary COP registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-71. COP_INIT

```
ioctl(COP, COP_INIT, NULL);
```

This code initializes the COP module by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.4.3.2 COP_DEVICE - enable/disable COP device

Call(s):

```
void ioctl(const int *pModuleBase, COP_DEVICE, bool param);
```

Arguments:

Table 5-82. COP_DEVICE ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Use COP_ENABLE to enable or COP_DISABLE to disable COP module.

Description: The *COP_DEVICE* ioctl command enables (COP_ENABLE) / disables (COP_DISABLE) the COP module. Before the COP module is used, the COP device should be configured (in disabled state) and then enabled. This command writes directly into the COP Control Register CEN ("COP Enable") bit.

Returns: None.

Range Issues: None

Special Issues: None.

Design/Implementation: The *COP_DEVICE* command is implemented as a macro.

Example 5-72. COP_DEVICE

```
/* configure desired COP parameters */
/* ... */
/* enable it */
ioctl(COP, COP_DEVICE, COP_ENABLE);
```

This code enables the COP module. Before enabling, the COP configuration should be taken.

5.4.3.3 COP_SET_TIMEOUT - sets COP timeout period

Call(s):

```
void ioctl(const int *pModuleBase, COP_SET_TIMEOUT,
           UWord16 param);
```

Arguments:

Table 5-83. COP_SET_TIMEOUT ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Parameter to select COP Timeout Divisor. Range 1-65535.

Description: The *COP_SET_TIMEOUT* ioctl command sets the 16-bit value of the COP Timeout Value. This command directly writes the COP Time-out Register. The param value is decremented before writing to the register.

$$\text{TimeoutPeriod} = \frac{\text{Timeout} \cdot \text{CopClock}}{\text{Prescaler}}$$

CopClock ... COP source clock. Typically an external clock [Hz]

Timeout ... COP timeout [sec]

Prescaler ... COP prescaler.

Returns: None.

Range Issues: None

Special Issues: The Prescaler constant can by on the MC56F800x **1/16/256/1024 and rest of the 56F800E family is 1024.**

Design/Implementation: The *COP_SET_TIMEOUT* ioctl command is implemented as a macro.

Example 5-73. COP_SET_TIMEOUT

```
/* set COP timeout to cca 1 sec for 8 MHz external clock */
ioctl(COP, COP_SET_TIMEOUT, 7813);
```

This code sets the COP timeout to cca 1 second.

5.4.3.4 COP_READ_COUNTER - read the COP counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase, COP_READ_COUNTER, NULL);
```

Arguments:

Table 5-84. COP_READ_COUNTER ioctl call arguments

pModuleBase	in	The COP module identifier. Use COP.

Description: The *COP_READ_COUNTER* ioctl command returns the current value of the COP counter as it counts down from the timeout value to zero. Count Register (COPCTR).

Returns: the Count Register (COPCTR) content.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_READ_COUNTER* ioctl command is implemented as a macro.

Example 5-74. COP_READ_COUNTER

```
UWord16 cnt = ioctl(COP, COP_READ_COUNTER, NULL);
```

This code reads the Count Register.

5.4.3.5 COP_CLEAR_COUNTER - clears COP timer

Call(s):

```
void ioctl(const int *pModuleBase, COP_CLEAR_COUNTER, NULL);
```

Arguments:

Table 5-85. COP_CLEAR_COUNTER ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.

Description: The *COP_CLEAR_COUNTER* ioctl command clears the COP counter by writing the service sequence (0x5555, 0xAAAA) to the COP Service Register. This command should be issued periodically before the COP timeout period elapses to prevent RESET. Usually it is inserted in the main program loop. It is not reasonable to insert it to interrupt routines, because interrupts may be enabled even when the program crashed and no code runaway may be detected.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_CLEAR_COUNTER* ioctl command is implemented as a macro.

Example 5-75. COP_CLEAR_COUNTER

```
ioctl(COP, COP_CLEAR_COUNTER, NULL);
```

This code clears the COP timer and prevents for generating a RESET. The command must be issued periodically with a period that is lower than the COP timeout period (see COP_SET_TIMEOUT).

5.4.3.6 COP_CLEAR_COUNTER_PART1 - first part of the COP timer clearing sequence

Call(s):

```
void ioctl(const int *pModuleBase, COP_CLEAR_COUNTER_PART1,
           NULL);
```

Arguments:

Table 5-86. COP_CLEAR_COUNTER_PART1 ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.

Description: The *COP_CLEAR_COUNTER_PART1* ioctl command writes the first part of the whole COP counter clearing service sequence (0x5555) to the COP Service Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_CLEAR_COUNTER_PART1* ioctl command is implemented as a macro.

Example 5-76. COP_CLEAR_COUNTER_PART1

```
ioctl(COP, COP_CLEAR_COUNTER_PART1, NULL);
```

This code writes 0x5555 to the COP Service Register.

5.4.3.7 COP_CLEAR_COUNTER_PART2 - second part of the COP timer clearing sequence

Call(s):

```
void ioctl(const int *pModuleBase, COP_CLEAR_COUNTER_PART2,
           NULL);
```

Arguments:

Table 5-87. COP_CLEAR_COUNTER_PART2 ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.

Description: The *COP_CLEAR_COUNTER_PART2* ioctl command writes the second part of the whole COP counter clearing service sequence (0xAAAA) to the COP Service Register

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_CLEAR_COUNTER_PART2* ioctl command is implemented as a macro.

Example 5-77. COP_CLEAR_COUNTER_PART2

```
ioctl(COP, COP_CLEAR_COUNTER_PART2, NULL);
```

This code writes 0xAAAA to the COP Service Register.

5.4.3.8 COP_RUN_IN_STOP - enable/disable COP device running in STOP mode

Call(s):

```
void ioctl(const int *pModuleBase, COP_RUN_IN_STOP, bool param);
```

Arguments:

Table 5-88. COP_RUN_IN_STOP ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Use COP_ENABLE to enable or COP_DISABLE to disable the COP module in STOP mode.

Description: The *COP_RUN_IN_STOP* ioctl command enables (COP_ENABLE) / disables (COP_DISABLE) the COP module in STOP mode. This command writes directly into the COP Control Register CSEN ("COP Stop Enable") bit.

Returns: None.

Range Issues: None

Special Issues: None.

Design/Implementation: The *COP_RUN_IN_STOP* command is implemented as a macro.

Example 5-78. COP_RUN_IN_STOP

```
ioctl(COP, COP_RUN_IN_STOP, COP_DISABLE);
```

This code disables the COP module in STOP (power down) mode.

5.4.3.9 COP_RUN_IN_WAIT - enable/disable COP device running in WAIT mode

Call(s):

```
void ioctl(const int *pModuleBase, COP_RUN_IN_WAIT, bool param);
```

Arguments:

Table 5-89. COP_RUN_IN_WAIT ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Use COP_ENABLE to enable or COP_DISABLE to disable COP module in STOP mode.

Description: The *COP_RUN_IN_WAIT* ioctl command enables (COP_ENABLE) / disables (COP_DISABLE) the COP module in STOP mode. This command writes directly in the COP Control Register CWEN ("COP Wait Enable") bit.

Returns: None.

Range Issues: None

Special Issues: None.

Design/Implementation: The *COP_RUN_IN_WAIT* command is implemented as a macro.

Example 5-79. COP_RUN_IN_WAIT

```
ioctl(COP, COP_RUN_IN_WAIT, COP_ENABLE);
```

This code enables the COP module to be active in WAIT (power down) mode.

5.4.3.10 COP_WRITE_PROTECT - write protects COP timer configuration

Call(s):

```
void ioctl(const int *pModuleBase, COP_WRITE_PROTECT, NULL);
```

Arguments:

Table 5-90. COP_WRITE_PROTECT ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Parameter not used.

Description: The *COP_WRITE_PROTECT* ioctl prevents changing the COP configuration by all previous commands, except the COP_CLEAR_COUNTER. It is intended to block any accidental change of the COP configuration by a runaway code. When this command is issued it is no longer possible to change the COP setting, until RESET.

This command writes directly in the COP Control Register CWP ("COP Write Protect") bit.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *COP_WRITE_PROTECT* ioctl command is implemented as a macro.

Example 5-80. COP_WRITE_PROTECT

```
ioctl(COP, COP_WRITE_PROTECT, NULL);
```

This code write protects all successive changes of the COP module, except for the COP_CLEAR_COUNTER command. Until RESET, change of the COP configuration is not possible.

5.4.3.11 COP_LOR_WATCHDOG - enable/disable loss-of-reference-clock watchdog

Call(s):

```
void ioctl(const int *pModuleBase, COP_LOR_WATCHDOG,
     UWord16 param);
```

Arguments:

Table 5-91. COP_LOR_WATCHDOG ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Use one of the following values: **COP_ENABLE** ... Enable loss-of-reference watchdog **COP_DISABLE** ... Disable loss-of-reference watchdog

Description: The *COP_LOR_WATCHDOG ioctl* command enables or disables the loss-of-reference-clock watchdog. Please see more details about this watchdog in the **MC56F802x/3x Peripheral Reference Manual**.

This command writes directly in the COP Control Register CLOREN ("COP Loss of Reference") bit.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *COP_LOR_WATCHDOG* ioctl command is implemented as a macro.

Example 5-81. COP_LOR_WATCHDOG

```
ioctl(COP, COP_LOR_WATCHDOG, COP_DISABLE);
```

This code disables the loss-of-reference-clock watchdog.

5.4.3.12 COP_SET_CLOCK_SOURCE - select COP clock source

Call(s):

```
void ioctl(const int *pModuleBase, COP_SET_CLOCK_SOURCE,
     UWord16 param);
```

Arguments:

Table 5-92. COP_SET_CLOCK_SOURCE ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP.
param	in	Use one of the following values: **on MC56F802x/3x** COP_MSTR_OSC ... MSTR_OSC clock is used to drive the COP COP_RLX_OSC ... Internal relaxation oscillator is used COP_IPBUS_CLK ... IPBUS clock is used to driver the COP (not-recommended) **on MC56F800x** COP_1KHZ ... 1kHz internal oscillator is used COP_IPBUS ... IPBUS clock is used to driver the COP COP_COSC ... Crystal oscillator is used COP_ROSC ... Internal relaxation oscillator is used

Description: The *COP_SET_CLOCK_SOURCE* ioctl command selects the COP clock source.

This command writes directly in the COP Control Register CLKSEL (“Clock Source Select”) bit.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x or MC56F800x.

Design/Implementation: The *COP_SET_CLOCK_SOURCE* ioctl command is implemented as a macro.

Example 5-82. COP_SET_CLOCK_SOURCE

```
ioctl(COP, COP_SET_CLOCK_SOURCE, COP_RLX_OSC);
```

This code selects internal relaxation oscillator as the COP clock source.

5.4.3.13 COP_SET_CLOCK_PRESCALER - select COP prescaler

Call(s):

```
void ioctl(const int *pModuleBase, COP_SET_CLOCK_PRESCALER,
     UWord16 param);
```

Arguments:

Table 5-93. COP_SET_CLOCK_PRESCALER ioctl call arguments

*pModuleBase	in	The COP module identifier. Use COP on MC56F800x.
param	in	Use one of the following values: COP_DIV1 ... set clock prescaler to 1 COP_DIV16 ... set clock prescaler to 16 COP_DIV256 ... set clock prescaler to 256 COP_DIV1024 ... set clock prescaler to 1024

Description: The *COP_SET_CLOCK_PRESCALER* ioctl command selects the COP clock prescaler.

This command writes directly in the COP Control Register PSS ("Prescaler Select") bits.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *COP_SET_CLOCK_PRESCALER* ioctl command is implemented as a macro.

Example 5-83. COP_SET_CLOCK_PRESCALER

```
ioctl(COP, COP_SET_CLOCK_PRESCALER, COP_DIV1024);
```

This code selects 1024 clock prescaler.

5.4.4 COP Driver Application

The COP driver application is designed for Freescale/Motorola EVM's and intended to illustrate the usage of this driver by a real example. It shows a most common way of using the COP module in user application.

The COP driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\cop_demo* directory and consists of the application project *cop.mcp* and the source code for the application *main.c*.

Example 5-84. COP Driver Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Wed, 07/Feb/2007, 10:19:17
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
----------------------------------------
    Core frequency: 8 MHz
    VCO frequency: 256 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Enable
    COP timeout: 2 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x2C1F
#define COP_COPCTL_INIT                  0x0002
#define COP_COPTO_INIT                   0x3D08

/*.
    SYS Configuration
```

```
----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
.*/
#define SIM_GPS_INIT                        0x0000

/*.
    SEMI Configuration
----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                    0x0005
#define SEMI_CSBAR1_INIT                    0x0005
#define SEMI_CSBAR2_INIT                    0x0005
#define SEMI_CSOR0_INIT                     0x1FC3
#define SEMI_CSOR1_INIT                     0x1BA3
#define SEMI_CSOR2_INIT                     0x1DA3

/*.
    INTC Configuration
----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
```

```
#define INTC_ICTL_INIT                  0x0000
#define INT_VECTOR_ADDR_1               cop_reset

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                 0x1FC1


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-85. COP Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application which demostrates how COP reset can be handled.
*              The application turns off the red and yellow LEDs and then let
*              COP counter to expire.
*
*              cop_reset turns on the red LED and then (after re-start) the main()
*              detects this source of reset and starts to toggle the yellow LED.
*
* TARGET: MC56F8xxx devices
*
*****************************************************************************/

#include "qs.h"

#include "sys.h"
#include "intc.h"
#include "gpio.h"
#include "cop.h"

/* board-specific LEDs */
#include "../board.h"

void Start();

/*
 *  Interrupt subroutine for COP (BEWARE: chip is in post-reset state)
 */

void cop_reset(void)
{
    /* RED LED turned on */
    ioctl(GPIO_LEDS, GPIO_SETAS_GPIO, LED_R);
    ioctl(GPIO_LEDS, GPIO_SETAS_OUTPUT, LED_R);
    ioctl(GPIO_LEDS, GPIO_SET_PIN, LED_R);

    /* perform startup actions to setup memory, variables, stack pointer etc.
       and call main() */
    asm { jmp >Start }
}

/*
 *  The main
 */

int main(void)
{
    /* init SYS module, see appconfig.h */
    ioctl(SYS, SYS_INIT, NULL);

    /* init GPIO port A for LEDs */
    ioctl(GPIO_LEDS, GPIO_SETAS_GPIO, LED_R | LED_Y);
    ioctl(GPIO_LEDS, GPIO_SETAS_OUTPUT, LED_R | LED_Y);

    /* last RESET was issued by COP, flash yellow led and loop forever */
```

```
    if (ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_COP_RESET))
    {
        /* clear COP reset status */
        ioctl(SYS, SYS_CLEAR_RESET_SOURCE, SYS_ALL_RESETS);

        for(;;)
        {
            ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_Y);
            archDelay(0xffff);
        }
    }

    /* RED LED turned off */
    ioctl(GPIO_LEDS, GPIO_CLEAR_PIN, LED_R | LED_Y);

    /* init COP module, see appconfig.h */
    ioctl(COP, COP_INIT, NULL);

    /* let COP expire */
    while(1)
    {
    }
}
```

5.5 SYS Driver

This section describes the API for the MC56F800E on-chip system support functions - system integration module, low voltage detection and external bus interface. The functionality of these modules itself is described in the device Data Sheet, **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.5.1 Introduction

The 56F800E devices have a reset circuitry which includes:

- Power-On Reset (POR) - keep chip in reset state until supply voltage rises up to a sufficient level
- External Reset - external RESET input
- COP (Computer Operating Properly) module Reset

SYS driver implements functions which detect the type of the previous Reset.

Beside the reset circuitry the MC56F83xx, MC56F801x and **MC56F802x/3x** devices have a Low Voltage Interrupt function (also referred as Power Supervisor or Brown-out detection). This function can generate an interrupt if the supply voltage drops below 2.2V or 2.7V thus maintaining reliable chip operation without additional external circuitry.

This section describes the SYS driver software providing the low level software layer interfacing hardware to/with software.

5.5.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.5.3.

Unlike the 56F800 processors where the three system modules were covered by the single peripheral address space, the 56F800E processors split the functionality into separate modules:

- SYS (SIM) - System Integration Module
- LVI - Low-Voltage Inhibit Module (Power Supervisor)
- SEMI (EMI)- System Bus External Memory Interface

Table 5-94. Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
SYS (SIM_BASE)	0xF240	0xF140	0xF100	0xF350
LVI (LVI_BASE)	N/A	0xF160	0xF140	0xF360
SEMI (SEMI_BASE)	N/A	N/A	N/A	0xF020

5.5.2.1 API Definition

The following header files are needed in order to use the SYS device driver:

Required Header File(s):

```
#include "qs.h"
#include "sys.h"
```

The following information may be found in the header file *sys.h*.

Public Data Structure(s):

None

5.5.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static SYS module configuration by the application startup code and driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Note that there are significant differences between SYS modules on MC56F83xx/MC56F801x and the newer MC56F802x/3x and MC56F800x devices.

Table 5-95. SYS Configuration Items for appconfig.h MC56F83xx/MC56F801x

SYMBOL	TYPE	DESCRIPTION
SIM_CONROL_INIT	UWord16	The initial value of the SIM Control Register.
SIM_PUDR_INIT	UWord16	The initial value of the SIM Pull-up Disable Register.
SIM_CLKOSR_INIT	UWord16	The initial value of the CLKO Select Register.
SIM_GPS_INIT	UWord16	The initial value of the GPIO Peripheral Select Register.
SIM_PCE_INIT	UWord16	The initial value of the Peripheral Clock Enable Register.
SIM_PCE2_INIT	UWord16	The initial value of the Peripheral Clock Enable Register 2.
LVI_CONTROL_INIT	UWord16	The initial value of the LVI Control Register.
SEMI_CSBARx_INIT (x=0..7)	UWord16	The initial values of the Chip-Select Base Address Registers (0-7). These values are written during the startup code.
SEMI_CSORx_INIT (x=0..7)	UWord16	The initial values of the Chip-Select Option Registers (0-7). These values are written during the startup code.
SEMI_CSTCx_INIT (x=0..7)	UWord16	The initial values of the Chip Select Timing Control Registers (0-7). These values are written during the startup code.
SEMI_BCR_INIT	UWord16	The initial value of the Bus Control Register. This value is written during the startup code.

Table 5-96. SYS Configuration Items for appconfig.h MC56F802x/3x

SYMBOL	TYPE	DESCRIPTION
SIM_CONROL_INIT	UWord16	The initial value of the SIM Control Register.
SIM_CLKOSR_INIT	UWord16	The initial value of the CLKO Select Register.
SIM_GPSA0_INIT SIM_GPSA1_INIT SIM_GPSB0_INIT SIM_GPSB1_INIT SIM_GPSCD_INIT	UWord16	The initial values of the GPIO Peripheral Select Registers for GPIO modules A-D.
SIM_IPS0_INIT SIM_IPS1_INIT SIM_IPS2_INIT	UWord16	The initial values of the Internal Peripheral Source Registers.
SIM_PCR_INIT	UWord16	The initial value of the Peripheral Clock Rate Register.
SIM_PCE0_INIT SIM_PCE1_INIT	Uword16	The initial values of the Peripheral Clock Enable Registers.
SIM_PSD0_INIT SIM_PSD1_INIT	Uword16	The initial values of the Peripheral Stop Disable Registers.
SIM_PROT_INIT	UWord16	The initial value of the Protection Register.
LVI_CONTROL_INIT	UWord16	The initial value of the LVI Control Register.

Table 5-97. SYS Configuration Items for appconfig.h MC56F800x

SYMBOL	TYPE	DESCRIPTION
SIM_CONROL_INIT	UWord16	The initial value of the SIM Control Register.
SIM_CLKOSR_INIT	UWord16	The initial value of the CLKO Select Register.
SIM_ISAL_INIT	UWord16	The initial value of the i/O shor Address Location Register.
SIM_GPSA_INIT SIM_GPSB0_INIT SIM_GPSB1_INIT SIM_GPSC_INIT SIM_GPSD_INIT	UWord16	The initial values of the GPIO Peripheral Select Registers for GPIO modules A-D.
SIM_IPS0_INIT SIM_IPS1_INIT	UWord16	The initial values of the Internal Peripheral Source Registers.
SIM_PCR_INIT	UWord16	The initial value of the Peripheral Clock Rate Register.
SIM_PCE_INIT	Uword16	The initial values of the Peripheral Clock Enable Registers.
SIM_SDR_INIT	Uword16	The initial values of the Peripheral Stop Disable Registers.
SIM_PROT_INIT	UWord16	The initial value of the Protection Register.

5.5.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* param);
```

Description: The *ioctl* call "changes" the modes of the SYS module or accesses the SYS register(s).

Arguments:

Table 5-98. SYS Driver Arguments - ioctl

pModuleBase	in	SYS module identifier. Use SYS. Power Supervisor module identifier. Use LVI. External Memory Interface module identifier. Use SEMI.
cmd	in	Commands found in *sys.h* which are used to modify the SYS/LVI/SEMI module status and control registers. See Table 5-99.
param	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-99. ioctl commands

Cmd	param	Return	Description
SYS_INIT	NULL	None	Initializes SYS (SIM and LVI) modules using the data from configuration file (appconfig.h).

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
SYS_STOP	SYS_ENABLE / SYS_DISABLE / **new on MC56F802x/3x and MC56F800x:** SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT	None	Enables or disables the STOP instruction.
SYS_STOP_PERMANENT_DISABLE (replaced by enhanced parameters of SYS_STOP command on MC56F802x/3x)	NULL	None	Permanently disables the STOP instruction. This state cannot be changed until Reset.
SYS_WAIT	SYS_ENABLE/ SYS_DISABLE **new on MC56F802x/3x and MC56F800x:** SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT	None	Enables or disables WAIT instruction.
SYS_WAIT_PERMANENT_DISABLE (replaced by enhanced parameters of SYS_STOP command on MC56F802x/3x)	NULL	None	Permanently disables the WAIT instruction. This state cannot be changed until Reset.
SYS_SOFTWARE_RESET	NULL	None	Issues software reset.
SYS_ONCE	SYS_ENABLE/ SYS_DISABLE	None	Enables or disables the OnCE module.
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_WRITE_SW_CONTROL_REG*n* (*n*=0..3)	UWord16	None	Writes a new value into the SIM Software Control Register.
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_READ_SW_CONTROL_REG*n* (*n*=0..3)	NULL	UWord16	Returns the content of the SIM Software Control Register.
SYS_READ_LSH_JTAG_ID	NULL	UWord16	Reads the Least Significant Half of JTAG ID.
SYS_READ_MSH_JTAG_ID	NULL	UWord16	Reads the Most Significant Half of JTAG ID.

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
SYS_TEST_RESET_SOURCE	SYS_SW_RESET | SYS_COP_RESET SYS_EXTERN_RESET | SYS_POWER_ON_RESET | **new on MC56F802x/3x:** SYS_COP_TOR_RESET | SYS_COP_LOR_RESET **new on MC56F800x:** SYS_COP_LOR_RESET | SYS_COP_CPU_RESET | SYS_PARTIAL_PD_RESET | SYS_LOW_VOLTAGE_RESET	None	Tests which event caused previous Reset.
SYS_CLEAR_RESET_SOURCE	SYS_SW_RESET | SYS_COP_RESET SYS_EXTERN_RESET | SYS_POWER_ON_RESET | **new on MC56F802x/3x:** SYS_COP_TOR_RESET | SYS_COP_LOR_RESET **new on MC56F800x:** SYS_COP_LOR_RESET | SYS_COP_CPU_RESET | SYS_PARTIAL_PD_RESET | SYS_LOW_VOLTAGE_RESET **all flags combined:** SYS_ALL_RESETS	None	Clears appropriate bits in the Reset Status Register.
MC56F83xx only: SYS_PULL_UP_ENABLE **MC56F83xx only:** SYS_PULL_UP_DISABLE	SYS_CAN_PINS | SYS_EMIMODE_PINS | SYS_RESET_PINS | SYS_IRQ_PINS | SYS_XBOOT_PINS | SYS_PWMB_PINS | SYS_PWMA0_PINS | SYS_PWMA1_PINS | SYS_DATA_PINS | SYS_CTRL_PINS | SYS_ADR_PINS | SYS_JTAG_PINS | SYS_TMRD_PINS | SYS_TMRC_PINS | SYS_TMRA_PINS	None	Enables or disables internal pull-ups for selected group of pins.
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_CLKOUT	SYS_ENABLE/ SYS_DISABLE	None	Enables or disables the CLKOUT processor pin.

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_CLKOUT_SELECT	SYS_CLKOUT_SYSCLK / SYS_CLKOUT_OSC / SYS_CLKOUT_FOUT / SYS_CLKOUT_ADCACLK / SYS_CLKOUT_ADCBCLK *list of available modes depends on the concrete device*	None	Selects which of internal clock signals is to be multiplexed on the CLKOUT processor pin.
MC56F800x only: SYS_CLKOUT_*ENABLE*	SYS_CLKO_0 | SYS_CLKO_1	None	Enables selected clock output
MC56F800x only: SYS_CLKOUT_*DISABLE*	SYS_CLKO_0 | SYS_CLKO_1	None	Disables selected clock output
MC56F800x only: SYS_SET_CLKOUT_0_SOURCE	SYS_CLKO_0_SYSCLK / SYS_CLKO_0_IPBCLK / SYS_CLKO_0_IPBCLK_3X / SYS_CLKO_0_MSTRCLK / SYS_CLKO_0_1KHZ / SYS_CLKO_0_RELAX_OSC / SYS_CLKO_0_CRYSTAL_OSC	None	Selects clock output 0 source.
MC56F800x only: SYS_SET_CLKOUT_1_SOURCE	SYS_CLKO_1_SYSCLK / SYS_CLKO_1_IPBCLK / SYS_CLKO_1_IPBCLK_3X / SYS_CLKO_1_MSTRCLK / SYS_CLKO_1_1KHZ / SYS_CLKO_0_RELAX_OSC / SYS_CLKO_0_CRYSTAL_OSC	None	Selects clock output 1source.
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_ACLK_ENABLE	SYS_ACLK_xxx *see the corresponding sys.h for the actual list of parameters*	None	Enables or disables replacement of various clock signals instead of address lines or GPIO signals.
MC56F83xx, MC56F801x and MC56F802x/3x only: SYS_ACLK_DISABLE			
MC56F83xx and MC56F801x only: SYS_SET_PADS_FUNCTION	Use SYS_PAD_xxx or SYS_PADS_xxx constants. *see the corresponding sys.h for the actual list of parameters*	None	Configures the pin multiplexer for a required operation mode.
MC56F801x only: SYS_ENABLE_INTERNAL_ TMR_SIGNAL	SYS_T3_PWMSYNC *list of timer signals which can be routed internally depends on the concrete device*	None	Enables internal interconnection between timer and other peripheral modules.
MC56F801x only: SYS_DISABLE_INTERNAL_ TMR_SIGNAL		None	Disables internal signal interconnection.

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
SYS_PERIPH_CLK_ENABLE	SYS_EMI_CLK | SYS_ADCB_CLK | SYS_ADCA_CLK | SYS_CAN_CLK | SYS_CAN2_CLK | SYS_DEC1_CLK | SYS_DEC0_CLK | SYS_TMRD_CLK | SYS_TMRC_CLK | SYS_TMRB_CLK | SYS_TMRA_CLK | SYS_SCI1_CLK | SYS_SCI0_CLK | SYS_SPI1_CLK | SYS_SPI0_CLK | SYS_PWMB_CLK | SYS_PWMA_CLK *list of peripherals available depends on the concrete device*	None	Enables clocks to the peripheral modules.
SYS_PERIPH_CLK_DISABLE		None	Disables clocks to the peripheral modules as a power savings feature.
SYS_WRITE_IO_SHORT_ADDR_ LOCATION_REG	UWord32	None	Writes the I/O Short Address Location Register.
SYS_READ_IO_SHORT_ADDR_ LOCATION_REG	NULL	UWord32	Reads the I/O Short Address Location Register.
MC56F80xx only: SYS_ENABLE_IN_STOP	SYS_T3_MOD | SYS_T2_MOD | SYS_T1_MOD | SYS_T0_MOD | SYS_SCI_MOD *list of peripherals available to operate in stop mode depends on the concrete device*	None	Selects modules which continue operation in the processor stop mode.
MC56F80xx only: SYS_DISABLE_IN_STOP		None	Selects modules for which the peripheral clock is to be disabled during the stop mode.
only on MC56F802x/3x: SYS_SET_TA1_SOURCE SYS_SET_TA2_SOURCE SYS_SET_TA3_SOURCE SYS_SET_PSRC0_SOURCE SYS_SET_PSRC1_SOURCE SYS_SET_PSRC2_SOURCE SYS_SET_FAULT1_SOURCE SYS_SET_FAULT2_SOURCE SYS_SET_DAC0SYNC_SOURCE SYS_SET_DAC1SYNC_SOURCE **general command format:** SYS_SET_*isig*_SOURCE *list of commands and possible parameters depends on the concrete device*	Each command accepts always one of the pre-defined values generally formatted as SYS_*isig*SRC_*osig* where *isig* is an internal input signal and *osig* is an internal output signal or “PIN”. For example: SYS_DAC1SYNCSRC_PIT0 SYS_FAULT2SRC_PIN	None	Each command configures internal source of one internal signal. The internal signal name is given by “isig” in the name of the signal, the possible sources are selected by a parameter value.

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
only on MC56F800x: SYS_SET_C0WS_SOURCE SYS_SET_C1WS_SOURCE SYS_SET_C2WS_SOURCE SYS_SET_T0_SOURCE SYS_SET_T1_SOURCE SYS_SET_PSRC0_SOURCE SYS_SET_PSRC1_SOURCE SYS_SET_PSRC2_SOURCE SYS_SET_FAULT1_SOURCE SYS_SET_FAULT2_SOURCE SYS_SET_FAULT3_SOURCE **eneral command format:** SYS_SET_*isig*_SOURCE *list of commands and possible parameters depends on the concrete device*	Each command accepts always one of the pre-defined values generally formatted as SYS_*isig*SRC_*osig* where *isig* is an internal input signal and *osig* is an internal output signal or “PIN”. For example: SYS_C0WS_T0 SYS_FAULT2SRC_PIN	None	Each command configures internal source of one internal signal. The internal signal name is given by “isig” in the name of the signal, the possible sources are selected by a parameter value.
MC56F802x/3x only: SYS_SET_A14PAD_FUNCTION SYS_SET_A13PAD_FUNCTION SYS_SET_A12PAD_FUNCTION SYS_SET_A11PAD_FUNCTION SYS_SET_A10PAD_FUNCTION SYS_SET_A9PAD_FUNCTION SYS_SET_A8PAD_FUNCTION SYS_SET_A6PAD_FUNCTION SYS_SET_A5PAD_FUNCTION SYS_SET_A4PAD_FUNCTION SYS_SET_B11PAD_FUNCTION SYS_SET_B10PAD_FUNCTION SYS_SET_B9PAD_FUNCTION SYS_SET_B8PAD_FUNCTION SYS_SET_B7PAD_FUNCTION SYS_SET_B6PAD_FUNCTION SYS_SET_B5PAD_FUNCTION SYS_SET_B4PAD_FUNCTION SYS_SET_B3PAD_FUNCTION SYS_SET_B2PAD_FUNCTION SYS_SET_B1PAD_FUNCTION SYS_SET_B0PAD_FUNCTION SYS_SET_C12PAD_FUNCTION SYS_SET_C8PAD_FUNCTION SYS_SET_D5PAD_FUNCTION **general command format:** SYS_SET_*pad*_FUNCTION *list of commands and possible parameters depends on the concrete device*	Each command accepts always one of the pre-defined values generally formatted as SYS_*pad*PAD_*sig* where *pad* is an external pin identifier in GPIO notation (e.g. A4, B4,..). And *sig* is an internal signal. For example: SYS_B3PAD_MOSI0 SYS_B3PAD_TA3 SYS_B4PAD_TA0 SYS_B4PAD_CLKO SYS_B4PAD_SS1 SYS_B4PAD_PSRC2 SYS_B4PAD_TB0	None	Each command configures one pin of the device and assigns a given peripheral function to it. The pin function takes effect when a pin is switched to peripheral mode using the GPIO_SETAS_PERIPHERAL command.

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
MC56F800x only: SYS_SET_A6PAD_FUNCTION SYS_SET_A5PAD_FUNCTION SYS_SET_A4PAD_FUNCTION SYS_SET_A3PAD_FUNCTION SYS_SET_B7PAD_FUNCTION SYS_SET_B6PAD_FUNCTION SYS_SET_B5PAD_FUNCTION SYS_SET_B4PAD_FUNCTION SYS_SET_B3PAD_FUNCTION SYS_SET_B2PAD_FUNCTION SYS_SET_B1PAD_FUNCTION SYS_SET_B0PAD_FUNCTION SYS_SET_C6PAD_FUNCTION SYS_SET_C0PAD_FUNCTION SYS_SET_D3PAD_FUNCTION SYS_SET_D2PAD_FUNCTION SYS_SET_D1PAD_FUNCTION SYS_SET_D0PAD_FUNCTION **general command format:** SYS_SET_*pad*_FUNCTION *list of commands and possible parameters depends on the concrete device*	Each command accepts always one of the pre-defined values generally formatted as SYS_*pad*PAD_*sig* where *pad* is an external pin identifier in GPIO notation (e.g. A4, B4,..). And *sig* is an internal signal. For example: SYS_B3PAD_MOSI SYS_B3PAD_TIN3 SYS_B3PAD_ANA3_ANB3 SYS_B3PAD_PWM5 SYS_B3PAD_CMP1_OUT SYS_B4PAD_T0 SYS_B4PAD_CLK_O SYS_B4PAD_MISO SYS_B4PAD_SDA SYS_B4PAD_RXD SYS_B4PAD_ANA0_ANB0	None	Each command configures one pin of the device and assigns a given peripheral function to it. The pin function takes effect when a pin is switched to peripheral mode using the GPIO_SETAS_PERIPHERAL command.
MC56F80xx only: SYS_HS_CLOCK_ENABLE	SYS_HS_TMR | SYS_HS_PWM *list of modules which can be sourced from the HS clock depends on the concrete device*	None	Enables high-speed clock (3x system clock typically) for the selected modules.
MC56F80xx only: SYS_HS_CLOCK_DISABLE		None	Disables high-speed clock for the selected modules.
MC56F801x and MC56F802x/3x only: SYS_SET_POWER_MODE	one of: SYS_NORMAL_POWER / SYS_REDUCED_POWER optionally combined with SYS_POWER_MODE_ PERMANENT	None	Controls the operation mode of the device's "Large Regulator". In some circumstances, the regulator may be put into reduced power mode. See the device data sheet for more details.
MC56F801x and MC56F802x/3x only: SYS_GET_POWER_MODE	NULL	UWord16	Gets current power mode. Return value can be tested for presence of the SYS_REDUCED_POWER and SYS_POWER_MODE_ PERMANENT flags.
MC56F802x/3x and MC56F800x only: SYS_WPROTECT_CLOCK_SETTINGS	SYS_ENABLE / SYS_DISABLE / SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT	None	Write-protects clock-related settings configured by the following commands: SYS_PERIPH_CLK_ENABLE SYS_ENABLE_IN_STOP SYS_HS_CLOCK_ENABLE

Table 5-99. ioctl commands (Continued)

Cmd	param	Return	Description
MC56F802x/3x and MC56F800x only: SYS_WPROTECT_SIGNALS_ROUTING	SYS_ENABLE / SYS_DISABLE / SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT	None	Write-protects signal-routing settings configured by the following commands: SYS_SET_*isig*_SOURCE SYS_SET_*pad*_FUNCTION
LVI commands: (only on MC56F83xx, MC56F801x and MC56F802x/3x)			
LVI_GET_LOW_VOLTAGE	LVI_22V_LEVEL | LVI_27V_LEVEL	UWord16	Reads value of the Low Voltage interrupt sticky flags in the Power Supervisor Status Register.
LVI_GET_NONSTICKY_INT_SOURCE	LVI_22V_LEVEL | LVI_27V_LEVEL	UWord16	Reads value of the Low Voltage interrupt non-sticky flags in the Power Supervisor Status Register.
LVI_CLEAR_LOW_VOLTAGE_INT	LVI_INT | LVI_22V_LEVEL | LVI_27V_LEVEL	None	Clears the Low Voltage interrupt flags in the Power Supervisor Status Register.
LVI_INT_ENABLE	LVI_22V_LEVEL | LVI_27V_LEVEL	None	Enables Low Voltage 2.2V, 2.7V detection interrupts.
LVI_INT_DISABLE	LVI_22V_LEVEL | LVI_27V_LEVEL	None	Disables Low Voltage 2.2V, 2.7V detection interrupts.
SEMI commands (selected MC56F83xx devices only):			
SEMI_SET_DRIVE_BUS *available on devices with external memory interface*	SEMI_DRIVEN/ SEMI_NON_DRIVEN	None	Controls the state of the external system bus when bus is not accessed. Use SEMI_DRIVEN to force bus pins to active state even when bus is not accessed or SEMI_NON_DRIVEN to leave bus tri-stated when not accessed (pull-ups may be active).
SEMI_WRITE_BASEREG*n* (*n*=0..7) SEMI_WRITE_OPTIONREG*n* (*n*=0..7) SEMI_WRITE_CONTROLREG *available on devices with external memory interface*	UWord16, value to be written into the external memory interface register	None	Direct write access to the SEMI registers. Note that in most cases it is sufficient to set up the SEMI module using static values from *appconfig.h*.
SEMI_READ_BASEREG*n* (*n*=0..7) SEMI_READ_OPTIONREG*n* (*n*=0..7) SEMI_READ_CONTROLREG *available on devices with external memory interface*	NULL	UWord16 (register value)	Direct read access to the SEMI registers.

5.5.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrates the usage of the ioctl commands.

5.5.3.1 SYS_INIT - initialize system support functions

Call(s):

```
void ioctl(const int *pModuleBase, SYS_INIT, NULL);
```

Arguments:

Table 5-100. SYS_INIT ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_INIT* ioctl command executes the initialization function, which initializes all configured SYS (SIM and LVI) registers by values defined in *appconfig.h*. It is intended for static configuration of the system modules after power-up. This command should be issued among the first calls in the *main()* routine.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_INIT* ioctl command is implemented as a function call. Only necessary SYS registers, which are defined in the *appconfig.h* are initialized. The execution time depends on the number of register values defined.

Example 5-86. SYS_INIT

```
ioctl(SYS, SYS_INIT, NULL);
```

This code initializes the system support registers by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.5.3.2 SYS_STOP - enables or disables the STOP mode

Call(s):

```
void ioctl(const int *pModuleBase, SYS_STOP, UWord16 param);
```

Arguments:

Table 5-101. SYS_STOP ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use SYS_ENABLE to enable or SYS_DISABLE to disable STOP mode. **new on MC56F802x/3x and MC56F800x:** SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT

Description: The *SYS_STOP* ioctl command enables or disables the STOP processor mode. When disabled, the STOP instructions act as a loop. This command directly accesses the appropriate bits in the SIM Control Register.

Two new "PERMANENT" parameter values can be used on MC56F802x/3x and MC56F800x to make the desired setting applied permanently until the next device reset.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_STOP* command is implemented as a macro.

Example 5-87. SYS_STOP

```
ioctl(SYS, SYS_STOP, SYS_DISABLE);
/* ... */
ioctl(SYS, SYS_STOP, SYS_ENABLE);
```

This code disables the STOP mode and then enables it again.

5.5.3.3 SYS_STOP_PERMANENT_DISABLE - permanently disables the STOP mode

Call(s):

```
void ioctl(const int *pModuleBase,
           SYS_STOP_PERMANENT_DISABLE, NULL);
```

Arguments:

Table 5-102. SYS_STOP_PERMANENT_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_STOP_PERMANENT_DISABLE* ioctl command behaves as the SYS_STOP command with SYS_DISABLE parameter. The difference is that the state cannot be changed by software until the Reset occurs. This command directly accesses the appropriate bit in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_STOP_PERMANENT_DISABLE* command is implemented as a macro.

Example 5-88. SYS_STOP_PERMANENT_DISABLE

```
ioctl(SYS, SYS_STOP_PERMANENT_DISABLE, NULL);
```

This code disables permanently the STOP mode. This state cannot be changed until reset.

5.5.3.4 SYS_WAIT - enables or disables the WAIT mode

Call(s):

```
void ioctl(const int *pModuleBase, SYS_WAIT, UWord16 param);
```

Arguments:

Table 5-103. SYS_WAIT ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use SYS_ENABLE to enable or SYS_DISABLE to disable WAIT mode. **new on MC56F802x/3x and MC56F800x:** SYS_ENABLE_PERMANENT / SYS_DISABLE_PERMANENT

Description: The *SYS_WAIT* ioctl command enables or disables the WAIT processor mode. When disabled, the WAIT instructions act as a loop. This command directly accesses the appropriate bits in the SIM Control Register.

Two new “PERMANENT” parameter values can be used on MC56F802x/3x and MC56F800x to make the desired setting applied permanently until the next device reset.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_WAIT* command is implemented as a macro.

Example 5-89. SYS_WAIT

```
ioctl(SYS, SYS_WAIT, SYS_DISABLE);
/* ... */
ioctl(SYS, SYS_WAIT, SYS_ENABLE);
```

This code disables the WAIT mode and then enables it again.

5.5.3.5 SYS_WAIT_PERMANENT_DISABLE - permanently disables the WAIT mode

Call(s):

```
void ioctl(const int *pModuleBase,
           SYS_WAIT_PERMANENT_DISABLE, NULL);
```

Arguments:

Table 5-104. SYS_WAIT_PERMANENT_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_WAIT_PERMANENT_DISABLE* ioctl command behaves as the SYS_WAIT command with SYS_DISABLE parameter. The difference is that the state cannot be changed by software until the Reset occurs. This command directly accesses the appropriate bit in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_WAIT_PERMANENT_DISABLE* command is implemented as a macro.

Example 5-90. SYS_WAIT_PERMANENT_DISABLE

```
ioctl(SYS, SYS_WAIT_PERMANENT_DISABLE, NULL);
```

This code disables permanently the WAIT mode. This state cannot be changed until reset.

5.5.3.6 SYS_SOFTWARE_RESET - issue software reset

Call(s):

```
void ioctl(const int *pModuleBase,
           SYS_SOFTWARE_RESET, NULL);
```

Arguments:

Table 5-105. SYS_SOFTWARE_RESET ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_SOFTWARE_RESET* ioctl issues software reset immediately.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_SOFTWARE_RESET* command is implemented as a macro.

Example 5-91. SYS_SOFTWARE_RESET

```
ioctl(SYS, SYS_SOFTWARE_RESET, NULL);
```

This code issues the software reset.

5.5.3.7 SYS_ONCE - enables or disables OnCE module

Call(s):

```
void ioctl(const int *pModuleBase, SYS_ONCE, UWord16 param);
```

Arguments:

Table 5-106. SYS_ONCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use SYS_ENABLE to enable or SYS_DISABLE to disable OnCE module.

Description: The *SYS_ONCE* ioctl command enables or disables the OnCE module.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SYS_ONCE* command is implemented as a macro.

Example 5-92. SYS_ONCE

```
ioctl(SYS, SYS_ONCE, SYS_DISABLE);
/* ... */
ioctl(SYS, SYS_ONCE, SYS_ENABLE);
```

This code disables the OnCE module and then enables it again.

5.5.3.8 SYS_WRITE_SW_CONTROL_REG*n* - writes SIM Software Control Register

Call(s):

```
void ioctl(const int *pModuleBase, SYS_WRITE_SW_CONTROL_REGn,
           UWord16 param);

      ... where n = 0..3
```

Arguments:

Table 5-107. SYS_WRITE_SW_CONTROL_REGn ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Value to be written to the register

Description: The *SYS_WRITE_SW_CONTROL_REGn* ioctl command writes specified value to one of the four SIM Software Control Registers.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The SYS_WRITE_SW_CONTROL_REGn command is implemented as a macro.

Example 5-93. SYS_WRITE_SW_CONTROL_REGn

```
ioctl(SYS, SYS_WRITE_SW_CONTROL_REG2, 0x0000);
```

This code writes 0x0000 to the SIM Software Control Register 2.

5.5.3.9 SYS_READ_SW_CONTROL_REG*n* - reads SIM Software Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SYS_READ_SW_CONTROL_REGn,
                NULL);

      ... where n = 0..3
```

Arguments:

Table 5-108. SYS_READ_SW_CONTROL_REGn ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_READ_SW_CONTROL_REGn* ioctl command reads the value of one of the four SIM Software Control Registers.

Returns: content of the selected SIM Software Control Register as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The SYS_READ_SW_CONTROL_REGn command is implemented as a macro.

Example 5-94. SYS_READ_SW_CONTROL_REGn

```
UWord16 tmp = ioctl(SYS, SYS_READ_SW_CONTROL_REG0, NULL);
```

This code reads the SIM Software Control Register 0.

5.5.3.10 SYS_READ_LSH_JTAG_ID - reads the Least Significant Half of JTAG ID

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SYS_READ_LSH_JTAG_ID,
              NULL);
```

Arguments:

Table 5-109. SYS_READ_LSH_JTAG_ID ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_READ_LSH_JTAG_ID* ioctl command reads the Least Significant Half of JTAG ID.

Returns: content of the Least Significant Half of JTAG ID Register as UWord16.

Range Issues: for the MC56F83xx and MC56F801x this reads 0x401D. For MC56F802x/3x this reads 0x801D

Special Issues: None.

Design/Implementation: The SYS_READ_LSH_JTAG_ID command is implemented as a macro.

Example 5-95. SYS_READ_LSH_JTAG_ID

```
UWord16 lID = ioctl(SYS, SYS_READ_LSH_JTAG_ID, NULL);
```

This code reads the Least Significant Half of JTAG ID.

5.5.3.11 SYS_READ_MSH_JTAG_ID - reads the Most Significant Half of JTAG ID

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SYS_READ_MSH_JTAG_ID,
              NULL);
```

Arguments:

Table 5-110. SYS_READ_MSH_JTAG_ID ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_READ_MSH_JTAG_ID* ioctl command reads the Most Significant Half of JTAG ID.

Returns: content of the Most Significant Half of JTAG ID Register as UWord16.

Range Issues: for the MC56F83xx this reads 0x01F4 and for MC56F80xx this reads 0x01F2.

Special Issues: None.

Design/Implementation: The SYS_READ_MSH_JTAG_ID command is implemented as a macro.

Example 5-96. SYS_READ_LSH_JTAG_ID

```
UWord16 mID = ioctl(SYS, SYS_READ_MSH_JTAG_ID, NULL);
```

This code reads the Most Significant Half of JTAG ID.

5.5.3.12 SYS_TEST_RESET_SOURCE - test the source of the previous Reset

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SYS_TEST_RESET_SOURCE,
          UWord16 param);
```

Arguments:

Table 5-111. SYS_TEST_RESET_SOURCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the type of Reset to test. Use one of or the combination of the predefined constants: SYS_SOFTWARE_RESET | SYS_COP_RESET | SYS_EXTERN_RESET | SYS_POWER_ON_RESET **new on MC56F802x/3x:** SYS_COP_TOR_RESET | SYS_COP_LOR_RESET **new on MC56F800x:** SYS_COP_LOR_RESET | SYS_COP_CPU_RESET | SYS_PARTIAL_PD_RESET | SYS_LOW_VOLTAGE_RESET

Description: The *SYS_TEST_RESET_SOURCE* ioctl command reads and tests the type of the previous Reset. Use predefined constants to define which Reset types to be tested. This command reads and tests directly the SIM Reset Status Register bits.

On the MC56F802x/3x devices, the new loss-of-reference-clock watchdog is implemented so there are two new reset-source values, one for standard COP reset (SYS_COP_TOR_RESET) and one for the loss-of-reference-clock COP reset (SYS_COP_LOR_RESET). For a backward compatibility, the SYS_COP_RESET is defined as a combination of the two reset sources.

On the MC56F800x devices there is test flag for standard COP reset (SYS_COP_CPU_RESET). Another COP flag is for lost-of-reference-clock COP reset (SYS_COP_LOR_RESET). There are new Partial Power Down flag (SYS_PARTIAL_PD_RESET) and Low Voltage Detect Reset (SYS_LOW_VOLTAGE_RESET)

Returns: True (non-zero) if at least one specified Reset type occurred. Zero if neither of those specified was active.

Range Issues: None.

Special Issues: The reset source bits in the SIM Reset Status Register are “sticky” and are not cleared automatically if the appropriate reset did not apply. Use the *SYS_CLEAR_RESET_SOURCE* to clear the reset status bits after you read it.

Design/Implementation: The *SYS_TEST_RESET_SOURCE* command is implemented as a macro.

Example 5-97. SYS_TEST_RESET_SOURCE

```
if (ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_COP_RESET))
{
/* previous RESET was caused by COP */
}
else
{
/* previous RESET was not caused by COM */
}
/* clear reset status bits for next use*/
ioctl (SYS, SYS_CLEAR_RESET_SOURCE, SYS_ALL_RESETS);
```

This code tests whether the previous Reset was caused by the COP module. Then it clears the reset status bits so the next Reset source can be identified correctly.

5.5.3.13 SYS_CLEAR_RESET_SOURCE - clear the reset status bits

Call(s):

```
void ioctl(const int *pModuleBase, SYS_CLEAR_RESET_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-112. SYS_CLEAR_RESET_SOURCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the Reset status bits to be cleared. Use one of or the combination of the predefined constants: SYS_SOFTWARE_RESET | SYS_COP_RESET | SYS_EXTERN_RESET | SYS_POWER_ON_RESET **new on MC56F802x/3x:** SYS_COP_TOR_RESET | SYS_COP_LOR_RESET **new on MC56F800x:** SYS_COP_LOR_RESET | SYS_COP_CPU_RESET | SYS_PARTIAL_PD_RESET | SYS_LOW_VOLTAGE_RESET Or use the SYS_ALL_RESETS, which is a combination of all constants above.

Description: The *SYS_CLEAR_RESET_SOURCE* ioctl command clears the specified bits in the reset status register. The reset source bits in the SIM Reset Status Register are “sticky” and are not cleared automatically if the appropriate reset did not apply. Use the *SYS_CLEAR_RESET_SOURCE* to clear the reset status bits after you read it with *SYS_TEST_RESET_SOURCE*.

On the MC56F802x/3x devices, the new loss-of-reference-clock watchdog is implemented so there are two new reset-source values, one for standard COP reset (SYS_COP_TOR_RESET) and one for the loss-of-reference-clock COP reset (SYS_COP_LOR_RESET). For a backward compatibility, the SYS_COP_RESET is defined as a combination of the two reset sources.

On the MC56F800x devices there is test flag for standard COP reset (SYS_COP_CPU_RESET). Another COP flag is for lost-of-reference-clock COP reset (SYS_COP_LOR_RESET). There are new Partial Power Down flag (SYS_PARTIAL_PD_RESET) and Low Voltage Detect Reset (SYS_LOW_VOLTAGE_RESET)

Returns: None.

Range Issues: None.

Special Issues:

Design/Implementation: The *SYS_CLEAR_RESET_SOURCE* command is implemented as a macro.

Example 5-98. SYS_CLEAR_RESET_SOURCE

```
if (ioctl(SYS, SYS_CLEAR_RESET_SOURCE, SYS_COP_RESET))
{
/* previous RESET was caused by COP */
}
else
{
/* previous RESET was not caused by COM */
}
/* clear reset status bits for next use*/
ioctl (SYS, SYS_CLEAR_RESET_SOURCE, SYS_ALL_RESETS);
```

This code tests whether the previous Reset was caused by the COP module. Then it clears the reset status bits so the next Reset source can be identified correctly.

5.5.3.14 SYS_PULL_UP_ENABLE - enables pull-up resistors for external bus and/or peripheral pins

Call(s):

```
void ioctl(const int *pModuleBase, SYS_PULL_UP_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-113. SYS_PULL_UP_ENABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the pin groups for which to enable the internal pull-up resistors. Use a combination (|) of the following constants: SYS_CAN_PINS | SYS_EMIMODE_PINS | SYS_RESET_PINS | SYS_IRQ_PINS | SYS_XBOOT_PINS | SYS_PWMB_PINS | SYS_PWMA0_PINS | SYS_PWMA1_PINS | SYS_DATA_PINS | SYS_CTRL_PINS | SYS_ADR_PINS | SYS_JTAG_PINS | SYS_TMRD_PINS | SYS_TMRC_PINS | SYS_TMRA_PINS. Note that not all parameters are available on all chips, see *sys.h*.

Description: The *SYS_PULL_UP_ENABLE* ioctl command enables the pull-up resistors for a chosen groups of pins. This command writes directly in the SIM Pull-up Disable Register bits.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx.

Design/Implementation: The *SYS_PULL_UP_ENABLE* command is implemented as a macro.

Example 5-99. SYS_PULL_UP_ENABLE

```
ioctl(SYS, SYS_PULL_UP_ENABLE, SYS_TMRA_PINS | SYS_ADR_PINS |
      SYS_DATA_PINS);
```

This code enables pull-ups for timer A pins, address bus pins and data bus pins.

5.5.3.15 SYS_PULL_UP_DISABLE - disables pull-up resistors for external bus and/or peripheral pins

Call(s):

```
void ioctl(const int *pModuleBase, SYS_PULL_UP_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-114. SYS_PULL_UP_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the pin groups for which to disable the internal pull-up resistors. Use a combination (|) of the following constants: SYS_CAN_PINS | SYS_EMIMODE_PINS | SYS_RESET_PINS | SYS_IRQ_PINS | SYS_XBOOT_PINS | SYS_PWMB_PINS | SYS_PWMA0_PINS | SYS_PWMA1_PINS | SYS_DATA_PINS | SYS_CTRL_PINS | SYS_ADR_PINS | SYS_JTAG_PINS | SYS_TMRD_PINS | SYS_TMRC_PINS | SYS_TMRA_PINS. Note that not all parameters are available on all chips, see *sys.h*.

Description: The *SYS_PULL_UP_DISABLE* ioctl command disables the pull-up resistors for a chosen groups of pins. This command writes directly in the SIM Pull-up Disable Register bits.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx.

Design/Implementation: The *SYS_PULL_UP_DISABLE* command is implemented as a macro.

Example 5-100. SYS_PULL_UP_DISABLE

```
ioctl(SYS, SYS_PULL_UP_ENABLE, SYS_TMRA_PINS | SYS_ADR_PINS |
      SYS_DATA_PINS);
```

This code disables pull-ups for timer A pins, address bus pins and data bus pins.

5.5.3.16 SYS_CLKOUT - enables or disables the CLKOUT pin

Call(s):

```
void ioctl(const int *pModuleBase, SYS_CLKOUT, UWord16 param);
```

Arguments:

Table 5-115. SYS_CLKOUT ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use SYS_ENABLE to enable or SYS_DISABLE to disable CLKOUT processor pin.

Description: The *SYS_CLKOUT* ioctl command enables or disables the CLKOUT pin. When enabled, the CLKOUT pin outputs the clock signals selected by the *SYS_CLKOUT_SELECT* command. When disabled, the CLKOUT pin is tri-stated.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *SYS_CLKOUT* command is implemented as a macro.

Example 5-101. SYS_CLKOUT

```
ioctl(SYS, SYS_CLKOUT, SYS_DISABLE);
/* ... */
ioctl(SYS, SYS_CLKOUT, SYS_ENABLE);
```

This code disables (tri-states) the CLKOUT processor pin and then enables it again.

5.5.3.17 SYS_CLKOUT_SELECT - selects the clock signal for the CLKOUT pin

Call(s):

```
void ioctl(const int *pModuleBase, SYS_CLKOUT_SELECT,
     UWord16 param);
```

Arguments:

Table 5-116. SYS_CLKOUT_SELECT ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use one of the following clock signal identifiers: SYS_CLKOUT_SYSCLK / SYS_CLKOUT_OSC / SYS_CLKOUT_FOUT / SYS_CLKOUT_ADCACLK / SYS_CLKOUT_ADCBCLK. Note that not all parameters are available on all chips, see *sys.h*.

Description: The *SYS_CLKOUT_SELECT* ioctl command selects the clock signal which is then routed to the CLKOUT processor pin. The CLKOUT pin must be also enabled by the SYS_CLKOUT command.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *SYS_CLKOUT_SELECT* command is implemented as a macro.

Example 5-102. SYS_CLKOUT_SELECT

```
ioctl(SYS, SYS_CLKOUT_SELECT, SYS_CLKOUT_SYSCLK);
/* ... */
ioctl(SYS, SYS_CLKOUT_SELECT, SYS_CLKOUT_OSC);
```

This code first selects the (default) system core clock to be output on the CLKOUT pin. Then it replaces it by the oscillator clock.

5.5.3.18 SYS_CLKOUT_ENABLE - enables selected the CLKOUT pins

Call(s):

```
void ioctl(const int *pModuleBase, SYS_CLKOUT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-117. SYS_CLKOUT_ENABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the CLKOUT pins. Use consolidation of these predefined constants: SYS_CLKO_0 | SYS_CLKO_1

Description: The *SYS_CLKOUT_ENABLE* ioctl command enables the CLKOUT pins. This comand can enable CLKOUT 0 pin, when SYS_CLKOUT_0 is used as parameter and CLKOUT 1 pin, when SYS_CLKOUT_1 is used as parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *SYS_CLKOUT_ENABLE* command is implemented as a macro.

Example 5-103. SYS_CLKOUT_ENABLE

```
ioctl(SYS, SYS_CLKOUT_ENABLE, SYS_CLKO_0);
```

This code enables the CLKOUT 0 processor pin.

5.5.3.19 SYS_CLKOUT_DISABLE - disables selected the CLKOUT pins

Call(s):

```
void ioctl(const int *pModuleBase, SYS_CLKOUT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-118. SYS_CLKOUT_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Parameter to select the CLKOUT pins. Use consolidation of these predefined constants: SYS_CLKO_0 | SYS_CLKO_1

Description: The *SYS_CLKOUT_DISABLE* ioctl command disables the CLKOUT pins. This comand can disable CLKOUT 0 pin, when SYS_CLKOUT_0 is used as parameter and CLKOUT 1 pin, when SYS_CLKOUT_1 is used as parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *SYS_CLKOUT_DISABLE* command is implemented as a macro.

Example 5-104. SYS_CLKOUT_DISABLE

```
ioctl(SYS, SYS_CLKOUT_DISABLE, SYS_CLKO_1);
```

This code enables the CLKOUT 1 processor pin.

5.5.3.20 SYS_SET_CLKOUT_0_SOURCE - selects the clock signal source for the CLKOUT 0 pin

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_CLKOUT_0_SOURCE,
     UWord16 param);
```

Arguments:

Table 5-119. SYS_SET_CLKOUT_0_SOURCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use one of the following clock signal identifiers: SYS_CLKOUT_SYSCLK / SYS_CLKOUT_IPBCLK / SYS_CLKOUT_IPBCLK_3X / SYS_CLKOUT_MSTRCLK / SYS_CLKOUT_1KHZ. / SYS_CLKO_0_RELAX_OSC / SYS_CLKO_0_CRYSTAL_OSC

Description: The *SYS_SET_CLKOUT_0_SOURCE* ioctl command selects the clock signal which is then routed to the CLKOUT 0 processor pin. The CLKOUT 0 pin must be also enabled by the SYS_CLKOUT_ENABLE command.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *SYS_SET_CLKOUT_0_SOURCE* command is implemented as a macro.

Example 5-105. SYS_SET_CLKOUT_0_SOURCE

```
ioctl(SYS, SYS_SET_CLKOUT_0_SOURCE, SYS_CLKOUT_SYSCLK);
/* ... */
ioctl(SYS, SYS_SET_CLKOUT_0_SOURCE, SYS_CLKOUT_COSC);
```

This code first selects the (default) system core clock to be output on the CLKOUT 0 pin. Then it replaces it by the crystal oscillator clock.

5.5.3.21 SYS_SET_CLKOUT_1_SOURCE - selects the clock signal source for the CLKOUT 1 pin

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_CLKOUT_1_SOURCE,
     UWord16 param);
```

Arguments:

Table 5-120. SYS_SET_CLKOUT_1_SOURCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use one of the following clock signal identifiers: SYS_CLKOUT_SYSCLK / SYS_CLKOUT_IPBCLK / SYS_CLKOUT_IPBCLK_3X / SYS_CLKOUT_MSTRCLK / SYS_CLKOUT_1KHZ. / SYS_CLKO_0_RELAX_OSC / SYS_CLKO_0_CRYSTAL_OSC

Description: The *SYS_SET_CLKOUT_1_SOURCE* ioctl command selects the clock signal which is then routed to the CLKOUT 1 processor pin. The CLKOUT 1 pin must be also enabled by the SYS_CLKOUT_ENABLE command.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *SYS_SET_CLKOUT_1_SOURCE* command is implemented as a macro.

Example 5-106. SYS_SET_CLKOUT_1_SOURCE

```
ioctl(SYS, SYS_SET_CLKOUT_1_SOURCE, SYS_CLKOUT_SYSCLK);
/* ... */
ioctl(SYS, SYS_SET_CLKOUT_1_SOURCE, SYS_CLKOUT_COSC);
```

This code first selects the (default) system core clock to be output on the CLKOUT 1 pin. Then it replaces it by the crystal oscillator clock.

5.5.3.22 SYS_ACLK_ENABLE - enable to multiplex out the clock signals

Call(s):

```
void ioctl(const int *pModuleBase, SYS_ACLK_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-121. SYS_ACLK_ENABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Specify the address pins which are to be switched to appropriate clock outputs. **MC56F83xx** SYS_ACLK_OSC | SYS_ACLK_SYSCLK2X | SYS_ACLK_SYSCLK | SYS_ACLK_PRESC. **MC56F80xx**: SYS_ACLK_OSC | SYS_ACLK_SYSCLK | SYS_ACLK_SYSCLK2x | SYS_ACLK_SYSCLK3x Note that not all parameters are available on all chips, see *sys.h*.

Description: The *SYS_ACLK_ENABLE* ioctl command selects the internal clocks to be multiplexed out on address/GPIO pins.

Returns: None.

Range Issues: None.

Special Issues: This command is not implemented on devices where clock-output multiplexed pins are not available and this command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *SYS_ACLK_ENABLE* command is implemented as a macro.

Example 5-107. SYS_ACLK_ENABLE

```
ioctl(SYS, SYS_ACLK_ENABLE, SYS_ACLK_OSC);
```

This code enables the oscillator clock to output pin.

5.5.3.23 SYS_ACLK_DISABLE - disable to multiplex out the clock signals

Call(s):

```
void ioctl(const int *pModuleBase, SYS_ACLK_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-122. SYS_ACLK_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Specify the address pins which are to be switched back to the address functionality. **MC56F83xx** SYS_ACLK_OSC | SYS_ACLK_SYSCLK2X | SYS_ACLK_SYSCLK | SYS_ACLK_PRESC. **MC56F80xx**: SYS_ACLK_OSC | SYS_ACLK_SYSCLK | SYS_ACLK_SYSCLK2x | SYS_ACLK_SYSCLK3x Note that not all parameters are available on all chips, see *sys.h*.

Description: The *SYS_ACLK_DISABLE* ioctl command selects the pins which are to be switched back to the original functionality (e.g. as address pin).

Returns: None.

Range Issues: None.

Special Issues: This command is not implemented on devices where clock-output multiplexed pins are not available and this command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *SYS_ACLK_DISABLE* command is implemented as a macro.

Example 5-108. SYS_ACLK_DISABLE

```
ioctl(SYS, SYS_ACLK_DISABLE, SYS_ACLK_OSC);
```

This code reverts the A23 pin back to the original A23 functionality.

5.5.3.24 SYS_SET_PADS_FUNCTION - multiplexes between different peripheral functions

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_PADS_FUNCTION,
           UWord32 param);
```

Arguments:

Table 5-123. SYS_SET_PADS_FUNCTION ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use e.g. SYS_PADS_SPI1 to set peripheral function of the appropriate pins as SPI1. Also, you can use the constants for individual pin function control like SYS_PAD_C0_SCLK1, SYS_PAD_C0_SCLK1, SYS_PAD_C1_TB1, and similar. See the *sys.h* for the applicable list of parameters for your chip.

Description: The *SYS_SET_PADS_FUNCTION* ioctl command sets the peripheral function of the selected pin or a pin group. The function of the pin or pins can then be activated using the *GPIO_SETAS_PERIPHERAL* command, i.e. by writing to the GPIO Peripheral Enable (GPIO_PER) register. The *SYS_SET_PADS_FUNCTION* command modifies the appropriate bits in the SIM Peripheral Select Register (SIM_GPS).

Returns: None.

Range Issues: None.

Special Issues: Consult the Data Sheet of the CPU you are using for more information about what peripheral function are multiplexed in the SIM module. The multiplexer is connected a different way for different members of the 56F800E family CPUs.

Design/Implementation: The SYS_SET_PADS_FUNCTION command is implemented as a macro.

Example 5-109. SYS_SET_PADS_FUNCTION

```
ioctl(SYS, SYS_SET_PADS_FUNCTION, SYS_PADS_SPI1);
```

This code sets the function of selected pads as SPI1.

5.5.3.25 SYS_ENABLE_INTERNAL_TMR_SIGNAL - select internal timer signal routing

Call(s):

```
void ioctl(const int *pModuleBase,
     SYS_ENABLE_INTERNAL_TMR_SIGNAL, UWord16 param);
```

Arguments:

Table 5-124. SYS_ENABLE_INTERNAL_TMR_SIGNAL ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select internal signal. MC56F801x: SYS_T3_PWMSYNC

Description: The *SYS_ENABLE_INTERNAL_TMR_SIGNAL* ioctl command enables internal routing of timer module input signals. The current MC56F801x devices enable internal routing of one signal only: the Timer channel 3 input sourced from PWM reload_sync signal.

This command sets the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented on MC56F801x devices only. See the SYS_SET_isig_SOURCE ioctl command for a more general replacement on MC56F802x/3x platform.

Design/Implementation: The SYS_ENABLE_INTERNAL_TMR_SIGNAL command is implemented as a macro.

Example 5-110. SYS_ENABLE_INTERNAL_TMR_SIGNAL

```
ioctl(SYS, SYS_ENABLE_INTERNAL_TMR_SIGNAL, SYS_T3_PWMSYNC);
```

This code enables the Timer channel 3 input to be sourced from PWM reload_sync signal.

5.5.3.26 SYS_DISABLE_INTERNAL_TMR_SIGNAL - select internal timer signal routing

Call(s):

```
void ioctl(const int *pModuleBase,
     SYS_DISABLE_INTERNAL_TMR_SIGNAL, UWord16 param);
```

Arguments:

Table 5-125. SYS_DISABLE_INTERNAL_TMR_SIGNAL ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select internal signal. MC56F801x: SYS_T3_PWMSYNC

Description: The *SYS_DISABLE_INTERNAL_TMR_SIGNAL* ioctl command disables internal routing of timer module input signals. See *SYS_ENABLE_INTERNAL_TMR_SIGNAL* command description for more details.

This command sets the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented on MC56F801x devices. See the SYS_SET_isig_SOURCE ioctl command for a more general replacement on MC56F802x/3x platform.

Design/Implementation: The SYS_DISABLE_INTERNAL_TMR_SIGNAL command is implemented as a macro.

Example 5-111. SYS_DISABLE_INTERNAL_TMR_SIGNAL

```
ioctl(SYS, SYS_DISABLE_INTERNAL_TMR_SIGNAL, SYS_T3_PWMSYNC);
```

This code disables the Timer channel 3 input to be sourced from PWM reload_sync signal.

5.5.3.27 SYS_PERIPH_CLK_ENABLE - enable clocks to the peripherals

Call(s):

```
void ioctl(const int *pModuleBase, SYS_PERIPH_CLK_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-126. SYS_PERIPH_CLK_ENABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to enable clocks to the indicated peripheral(s). SYS_EMI_CLK | SYS_ADCB_CLK | SYS_ADCA_CLK | SYS_CAN_CLK | SYS_CAN2_CLK | SYS_DEC1_CLK | SYS_DEC0_CLK | SYS_TMRD_CLK | SYS_TMRC_CLK | SYS_TMRB_CLK | SYS_TMRA_CLK | SYS_SCI1_CLK | SYS_SCI0_CLK | SYS_SPI1_CLK | SYS_SPI0_CLK | SYS_PWMB_CLK | SYS_PWMA_CLK. Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_PERIPH_CLK_ENABLE* ioctl command enables clocks to the indicated peripherals. This command sets the appropriate bit(s) in the Peripheral Clock Enable Registers SIM_PCE and SIM_PCE2 (on some devices only).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SYS_PERIPH_CLK_ENABLE command is implemented as a macro.

Example 5-112. SYS_PERIPH_CLK_ENABLE

```
ioctl(SYS, SYS_PERIPH_CLK_ENABLE, SYS_PWMB_CLK | SYS_PWMA_CLK);
```

This code enables clocks to the PWM A and PWM B peripheral modules.

5.5.3.28 SYS_PERIPH_CLK_DISABLE - disable clocks to the peripherals

Call(s):

```
void ioctl(const int *pModuleBase, SYS_PERIPH_CLK_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-127. SYS_PERIPH_CLK_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to disable clocks to the indicated peripheral(s). SYS_EMI_CLK | SYS_ADCB_CLK | SYS_ADCA_CLK | SYS_CAN_CLK | SYS_DEC1_CLK | SYS_DEC0_CLK | SYS_TMRD_CLK | SYS_TMRC_CLK | SYS_TMRB_CLK | SYS_TMRA_CLK | SYS_SCI1_CLK | SYS_SCI0_CLK | SYS_SPI1_CLK | SYS_SPI0_CLK | SYS_PWMB_CLK | SYS_PWMA_CLK. Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_PERIPH_CLK_DISABLE* ioctl command disables clocks to the indicated peripherals as a power savings feature. This command clears the appropriate bit(s) in the Peripheral Clock Enable Registers SIM_PCE and SIM_PCE2 (on some devices only).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SYS_PERIPH_CLK_DISABLE command is implemented as a macro.

Example 5-113. SYS_PERIPH_CLK_DISABLE

```
ioctl(SYS, SYS_PERIPH_CLK_DISABLE, SYS_ADCA_CLK | SYS_ADCB_CLK);
```

This code disables clocks to the ADC A and ADC B peripheral modules.

5.5.3.29 SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG - writes the I/O Short Address Location Register

Call(s):

```
void ioctl(const int *pModuleBase,
           SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG, UWord32 param);
```

Arguments:

Table 5-128. SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Value to be written to the register

Description: The *SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG* ioctl command writes the specified value to the I/O Short Address Location Register (SIM_ISALH and SIM_ISALL).

Returns: None.

Range Issues: See the **User's Manual** for the details.

Special Issues: None.

Design/Implementation: The SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG command is implemented as a macro.

Example 5-114. SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG

```
ioctl(SYS, SYS_WRITE_IO_SHORT_ADDR_LOCATION_REG, 0x11111111);
```

This code writes 0x11111111 to the I/O Short Address Location Register.

5.5.3.30 SYS_READ_IO_SHORT_ADDR_LOCATION_REG - reads the I/O Short Address Location Register

Call(s):

```
UWord32 ioctl(const int *pModuleBase,
                SYS_READ_IO_SHORT_ADDR_LOCATION_REG, NULL);
```

Arguments:

Table 5-129. SYS_READ_IO_SHORT_ADDR_LOCATION_REG ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_READ_IO_SHORT_ADDR_LOCATION_REG* ioctl command reads the content of the I/O Short Address Location Register (SIM_ISALH and SIM_ISALL).

Returns: content of the I/O Short Address Location Register as UWord32.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SYS_READ_IO_SHORT_ADDR_LOCATION_REG command is implemented as a macro.

Example 5-115. SYS_READ_IO_SHORT_ADDR_LOCATION_REG

```
UWord32 tmp = ioctl(SYS, SYS_READ_IO_SHORT_ADDR_LOCATION_REG,
                    NULL);
```

This code reads the I/O Short Address Location Register.

5.5.3.31 SYS_ENABLE_IN_STOP - enable peripherals to run in stop mode

Call(s):

```
void ioctl(const int *pModuleBase, SYS_ENABLE_IN_STOP,
           UWord16 param);
```

Arguments:

Table 5-130. SYS_ENABLE_IN_STOP ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select peripheral modules. **MC56F80xx:** SYS_T3_MOD | SYS_T2_MOD | SYS_T1_MOD | SYS_T0_MOD | SYS_SCI_MOD Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_ENABLE_IN_STOP* ioctl command selects modules which continue operation during the processor stop mode. The peripheral clock of the modules selected by this command is not shut down when processor enters the stop mode. The peripherals may generate an interrupt to recover processor from the stop mode.

This command sets the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented only on devices with “run-in-stop” feature (MC56F80xx).

Design/Implementation: The SYS_ENABLE_IN_STOP command is implemented as a macro.

Example 5-116. SYS_ENABLE_IN_STOP

```
ioctl(SYS, SYS_ENABLE_IN_STOP, SYS_T0_MOD | SYS_SCI_MOD);
```

This code enables the Timer 0 and the SCI to run during the processor stop mode.

5.5.3.32 SYS_DISABLE_IN_STOP - disable peripherals to run in stop mode

Call(s):

```
void ioctl(const int *pModuleBase, SYS_DISABLE_IN_STOP,
           UWord16 param);
```

Arguments:

Table 5-131. SYS_DISABLE_IN_STOP ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select peripheral modules. **MC56F80xx:** SYS_T3_MOD | SYS_T2_MOD | SYS_T1_MOD | SYS_T0_MOD | SYS_SCI_MOD Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_DISABLE_IN_STOP* ioctl command un-selects modules to operate during the processor stop mode. The modules can be selected for such an operation using the *SYS_ENABLE_IN_STOP* command.

This command clears the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented only on devices with “run-in-stop” feature (MC56F80xx).

Design/Implementation: The SYS_DISABLE_IN_STOP command is implemented as a macro.

Example 5-117. SYS_DISABLE_IN_STOP

```
ioctl(SYS, SYS_DISABLE_IN_STOP, SYS_SCI_MOD);
```

This code configures the SCI peripheral clock to be stopped during the processor stop mode.

5.5.3.33 SYS_SET_*isig*_SOURCE - select internal signal routing

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_isig_SOURCE,
     UWord16 param);

command examples:

     On MC56F802x/3x:
     SYS_SET_TA1_SOURCE
     SYS_SET_TA2_SOURCE
     SYS_SET_TA3_SOURCE
     SYS_SET_PSRC0_SOURCE
     SYS_SET_PSRC1_SOURCE
     SYS_SET_PSRC2_SOURCE
     SYS_SET_FAULT1_SOURCE
     SYS_SET_FAULT2_SOURCE
     SYS_SET_DAC0SYNC_SOURCE
     SYS_SET_DAC1SYNC_SOURCE

     On MC56F800x:
     SYS_SET_C0WS_SOURCE
     SYS_SET_C1WS_SOURCE
     SYS_SET_C2WS_SOURCE
     SYS_SET_T0_SOURCE
     SYS_SET_T1_SOURCE
     SYS_SET_PSRC0_SOURCE
     SYS_SET_PSRC1_SOURCE
     SYS_SET_PSRC2_SOURCE
     SYS_SET_FAULT1_SOURCE
     SYS_SET_FAULT2_SOURCE
     SYS_SET_FAULT3_SOURCE
```

Arguments:

Table 5-132. SYS_SET_*isig*_SOURCE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Each command accepts always one of the pre-defined values generally formatted as: SYS_*isig*SRC_*osig* where *isig* is an internal input signal and must match the *isig* part in the command name. The *osig* is internal output signal or “PIN”. See the “sys.h” low-level driver header file for the list of all applicable values. For example: SYS_DAC1SYNCSRC_PIT0 SYS_FAULT2SRC_PIN

Description: The *SYS_SET_isig_SOURCE* ioctl command establishes internal routing of the selected signals. The signal is identified in the command name as “isig”. The source for the signal is selected by a proper parameter value.

This command sets the appropriate bit(s) in the SIM Internal Peripheral Source Register.

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F802x/3x and MC56F800x..

Design/Implementation: The SYS_SET_isig_SOURCE command is implemented as a macro.

Example 5-118. SYS_SET_isig_SOURCE

```
ioctl(SYS, SYS_SET_FAULT1_SOURCE, SYS_FAULT1SRC_COUTA_A);
```

This code selects that FAULT1 signal is sourced from the Analog Comparator A (CMP_A) asynchronous output.

5.5.3.34 SYS_SET_*pad*_FUNCTION - select device pad function

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_pad_FUNCTION,
     UWord16 param);

command examples:

     On MC56F802x/3x:
     SYS_SET_A14PAD_FUNCTION
     SYS_SET_A13PAD_FUNCTION
     SYS_SET_A12PAD_FUNCTION
     SYS_SET_A11PAD_FUNCTION
     SYS_SET_A10PAD_FUNCTION
     SYS_SET_A9PAD_FUNCTION
     SYS_SET_A8PAD_FUNCTION
     SYS_SET_A6PAD_FUNCTION
     SYS_SET_A5PAD_FUNCTION
     SYS_SET_A4PAD_FUNCTION

     SYS_SET_B11PAD_FUNCTION
     SYS_SET_B10PAD_FUNCTION
     SYS_SET_B9PAD_FUNCTION
     SYS_SET_B8PAD_FUNCTION
     SYS_SET_B7PAD_FUNCTION
     SYS_SET_B6PAD_FUNCTION
     SYS_SET_B5PAD_FUNCTION
     SYS_SET_B4PAD_FUNCTION
     SYS_SET_B3PAD_FUNCTION
     SYS_SET_B2PAD_FUNCTION
     SYS_SET_B1PAD_FUNCTION
     SYS_SET_B0PAD_FUNCTION

     SYS_SET_C12PAD_FUNCTION
     SYS_SET_C8PAD_FUNCTION

     SYS_SET_D5PAD_FUNCTION

     On MC56F800x:
     SYS_SET_A6PAD_FUNCTION
     SYS_SET_A5PAD_FUNCTION
     SYS_SET_A4PAD_FUNCTION
     SYS_SET_A3PAD_FUNCTION

     SYS_SET_B7PAD_FUNCTION
     SYS_SET_B6PAD_FUNCTION
     SYS_SET_B5PAD_FUNCTION
     SYS_SET_B4PAD_FUNCTION
     SYS_SET_B3PAD_FUNCTION
     SYS_SET_B2PAD_FUNCTION
     SYS_SET_B1PAD_FUNCTION
     SYS_SET_B0PAD_FUNCTION

     SYS_SET_C6PAD_FUNCTION
     SYS_SET_C0PAD_FUNCTION
```

```
SYS_SET_D3PAD_FUNCTION
SYS_SET_D2PAD_FUNCTION
SYS_SET_D1PAD_FUNCTION
SYS_SET_D0PAD_FUNCTION
```

Arguments:

Table 5-133. SYS_SET_*pad*_FUNCTION ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Each command accepts always one of the pre-defined values generally formatted as: SYS_*pad*PAD_*sig* where *pad* is an external pin identifier in GPIO notation (e.g. A4, B4, ..) and must match the *pad* part of the command name. The *sig* is an internal signal name. For example: SYS_B3PAD_MOSI0 SYS_B3PAD_TA3

Description: The *SYS_SET_pad_FUNCTION* ioctl command selects the peripheral function of the selected device pin. This pin still needs to be switched from the GPIO mode by using the GPIO_SETAS_PERIPHERAL command to achieve the desired functionality.

This command sets the appropriate bit(s) in one of SIM GPIO Peripheral Select Registers.

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F802x/3x and MC56F800x..

Design/Implementation: The SYS_SET_pad_FUNCTION command is implemented as a macro.

Example 5-119. SYS_SET_pad_FUNCTION

```
ioctl(SYS, SYS_SET_B4PAD_FUNCTION, SYS_B4PAD_CLKO);
```

This code selects that CLKOUT signal will be available as a peripheral function of the GPIO_B4 pin.

5.5.3.35 SYS_HS_CLOCK_ENABLE - select modules to use the HS clock

Call(s):

```
void ioctl(const int *pModuleBase, SYS_HS_CLOCK_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-134. SYS_HS_CLOCK_ENABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select peripheral modules. **MC56F801x:** SYS_HS_TMR | SYS_HS_PWM **MC56F802x/3x:** SYS_HS_TMRA | SYS_HS_TMRB | SYS_HS_PWM | SYS_HS_I2C Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_HS_CLOCK_ENABLE* ioctl command enables the selected modules to use the HS_PERF clock, which is typically running at 3x system clock (the ratio is different when not using the PLL clock).

This command sets the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented on **MC56F80xx** devices only.

Design/Implementation: The SYS_HS_CLOCK_ENABLE command is implemented as a macro.

Example 5-120. SYS_HS_CLOCK_ENABLE

```
ioctl(SYS, SYS_HS_CLOCK_ENABLE, SYS_HS_PWM);
```

This code enables the PWM clock to be sourced from HS_PERF clock.

5.5.3.36 SYS_HS_CLOCK_DISABLE - un-select modules to use the HS clock

Call(s):

```
void ioctl(const int *pModuleBase, SYS_HS_CLOCK_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-135. SYS_HS_CLOCK_DISABLE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the predefined constants to select peripheral modules. **MC56F801x:** SYS_HS_TMR | SYS_HS_PWM **MC56F802x/3x:** SYS_HS_TMRA | SYS_HS_TMRB | SYS_HS_PWM | SYS_HS_I2C Note that not all parameters are available on all chips and also some other parameters may be defined for a particlular device. For a complete list see *sys.h*.

Description: The *SYS_HS_CLOCK_DISABLE* ioctl command disables the HS_PERF clock routing to selected modules. See more details at *SYS_HS_CLOCK_ENABLE* command description.

This command sets the appropriate bit(s) in the SIM Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is implemented on **MC56F80xx** devices only.

Design/Implementation: The SYS_HS_CLOCK_DISABLE command is implemented as a macro.

Example 5-121. SYS_HS_CLOCK_DISABLE

```
ioctl(SYS, SYS_HS_CLOCK_DISABLE, SYS_HS_PWM);
```

This code reverts the PWM clock to be sourced from original peripheral clock again.

5.5.3.37 SYS_SET_POWER_MODE - set normal or reduced power mode

Call(s):

```
void ioctl(const int *pModuleBase, SYS_SET_POWER_MODE,
     UWord16 param);
```

Arguments:

Table 5-136. SYS_SET_POWER_MODE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use the one of the predefined constants:. SYS_NORMAL_POWER / SYS_REDUCED_POWER optionally combined with the flag: SYS_POWER_MODE_PERMANENT

Description: The *SYS_SET_POWER_MODE* ioctl command puts the internal voltage regulator to normal or reduced power mode. Using the *SYS_POWER_MODE_PERMANENT* flag logically or-ed with desired power mode, the change can be made permanent by write-protecting the power control register until next reset.

This command sets the appropriate bit(s) in the SIM Power Control Register (SIM_POWER).

Returns: None.

Range Issues: This commands are applicable only on MC56F801x and MC56F802x/3x..

Special Issues: This command is implemented on **MC56F80xx** devices only. Note that there are some pre-requisite actions necessary to put the processor to reduced power mode (switching to internal oscillator, turning off the PLL, and putting the oscillator to stand-by mode).

Design/Implementation: The SYS_SET_POWER_MODE command is implemented as a macro.

Example 5-122. SYS_SET_POWER_MODE

```
... /* reduced power mode preparation */
...
ioctl(SYS, SYS_SET_POWER_MODE, SYS_REDUCED_POWER);
```

This code finishes the sequence to enter the reduced-power (standby) mode.

5.5.3.38 SYS_GET_POWER_MODE - get actual power mode settings

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SYS_GET_POWER_MODE, NULL);
```

Arguments:

Table 5-137. SYS_GET_POWER_MODE ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.

Description: The *SYS_GET_POWER_MODE* ioctl command reads the power-control bits in the SIM Power Control Register (SIM_POWER).

Returns: The returned value may be compared with values used in *SYS_SET_POWER_MODE* command or can be tested for occurrence of the *SYS_REDUCED_POWER* or *SYS_POWER_MODE_PERMANENT* bits.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F801x and MC56F802x/3x.

Design/Implementation: The SYS_GET_POWER_MODE command is implemented as a macro.

Example 5-123. SYS_GET_POWER_MODE

```
if(ioctl(SYS, SYS_GET_POWER_MODE, NULL) &
      SYS_POWER_MODE_PERMANENT))
{
      /* power mode is set permanently until reset */
      ...
}
```

This code tests whether the power mode is write-protected.

5.5.3.39 SYS_WPROTECT_CLOCK_SETTINGS - protect clock-related settings

Call(s):

```
void ioctl(const int *pModuleBase, SYS_WPROTECT_CLOCK_SETTINGS,
     UWord16 param);
```

Arguments:

Table 5-138. SYS_WPROTECT_CLOCK_SETTINGS ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use one of the following constants **SYS_ENABLE** ... enable protection (may be changed later) **SYS_ENABLE_PERMANENT** ... enable protection until reset **SYS_DISABLE** ... disable protection (may be re-enabled later) **SYS_DISABLE_PERMANENT** ... disable protection until reset

Description: The *SYS_WPROTECT_CLOCK_SETTINGS* ioctl command write-protects the clock-related configuration bits in the SIM module. Depending on the parameter value, the protection may be activated or deactivated permanently (until the next reset).

The SYS_WPROTECT_CLOCK_SETTINGS command write-protects the PCEn, SDn, and PCR registers.

This command writes directly to the PCEP bit-field of the SIM Protection Register.

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F802x/3x and MC56F800x..

Design/Implementation: The SYS_WPROTECT_CLOCK_SETTINGS command is implemented as a macro.

Example 5-124. SYS_WPROTECT_CLOCK_SETTINGS

```
ioctl(SYS, SYS_WPROTECT_CLOCK_SETTINGS, SYS_ENABLE_PERMANENT);
```

This code write-protects the clock settings. It will not be possible to disable this protection until the next reset occurs.

5.5.3.40 SYS_WPROTECT_SIGNALS_ROUTING - protect signal-routing settings

Call(s):

```
void ioctl(const int *pModuleBase,
     SYS_WPROTECT_SIGNALS_ROUTING, UWord16 param);
```

Arguments:

Table 5-139. SYS_WPROTECT_SIGNALS_ROUTING ioctl call arguments

*pModuleBase	in	The System Integration Module identifier. Use SYS.
param	in	Use one of the following constants **SYS_ENABLE** ... enable protection (may be changed later) **SYS_ENABLE_PERMANENT** ... enable protection until reset **SYS_DISABLE** ... disable protection (may be re-enabled later) **SYS_DISABLE_PERMANENT** ... disable protection until reset

Description: The *SYS_WPROTECT_SIGNALS_ROUTING* ioctl command write-protects the signal-routing-related configuration bits in the SIM module. Depending on the parameter value, the protection may be activated or deactivated permanently (until the next reset).

The SYS_WPROTECT_SIGNALS_ROUTING command write-protects the GPSn and IPSn registers in the SIM module and also all GPIOx_PEREN, GPIOx_PPOUTM and GPIOx_DRIVE registers in GPIO modules.

This command writes directly to the GIPSP bit-field of the SIM Protection Register.

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F802x/3x and MC56F800x..

Design/Implementation: The SYS_WPROTECT_SIGNALS_ROUTING command is implemented as a macro.

Example 5-125. SYS_WPROTECT_SIGNALS_ROUTING

```
ioctl(SYS, SYS_WPROTECT_SIGNALS_ROUTING, SYS_DISABLE_PERMANENT);
```

This code disables write-protection of the signal settings. It will not be possible to enable this protection until the next reset occurs.

5.5.3.41 LVI_GET_LOW_VOLTAGE - reads low voltage sticky interrupt flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, LVI_GET_LOW_VOLTAGE,
          UWord16 param);
```

Arguments:

Table 5-140. LVI_GET_LOW_VOLTAGE ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the level of voltage threshold to test. Use one of the predefined constants or a combination of more constants: LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_GET_LOW_VOLTAGE* ioctl command reads and tests if the supply voltage dropped below the low voltage interrupt levels. Use predefined constants to define which level is to be tested - LVI_22V_LEVEL for 2.2V, LVI_27V_LEVEL for 2.7V. This command reads and tests directly the Power Supervisor Status Register sticky bits - LVIS27S (Bit 3) and LVIS22S (Bit 2).

Returns: True (non-zero) if the supply voltage dropped below at any of specified levels. Zero if the supply voltage remained above the specified level(s).

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x..

Design/Implementation: The *LVI_GET_LOW_VOLTAGE* command is implemented as a macro.

Example 5-126. LVI_GET_LOW_VOLTAGE

```
if (ioctl(LVI, LVI_GET_LOW_VOLTAGE, LVI_27V_LEVEL))
{
      /* supply voltage dropped below 2.7 V */
      /* ... */
      /* clear interrupt flags */
      ioctl(LVI, LVI_CLEAR_LOW_VOLTAGE_INT, LVI_27V_LEVEL);
}
else
{
      /* supply voltage is OK (above 2.7 V) */
}
```

This code tests if the supply voltage dropped under 2.7 V any time after it was previously tested.

5.5.3.42 LVI_GET_NONSTICKY_INT_SOURCE - reads low voltage non-sticky interrupt flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
          LVI_GET_NONSTICKY_INT_SOURCE, UWord16 param);
```

Arguments:

Table 5-141. LVI_GET_NONSTICKY_INT_SOURCE ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the level of voltage threshold to test. Use one of the predefined constants or a combination of more constants: LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_GET_NONSTICKY_INT_SOURCE* ioctl command reads and tests if the supply voltage dropped below the low voltage interrupt levels. Use predefined constants to define which level is to be tested - LVI_22V_LEVEL for 2.2V, LVI_27V_LEVEL for 2.7V. This command reads and tests directly the Power Supervisor Status Register non-sticky bits - LVIS27 (Bit 1) and LVIS22 (Bit 0).

Returns: True (non-zero) if the supply voltage dropped below at any of specified levels. Zero if the supply voltage remained above the specified level(s).

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The LVI_GET_NONSTICKY_INT_SOURCE command is implemented as a macro.

Example 5-127. LVI_GET_NONSTICKY_INT_SOURCE

```
if (ioctl(LVI, LVI_GET_NONSTICKY_INT_SOURCE, LVI_27V_LEVEL))
{
     /* supply voltage dropped below 2.7 V */
     /* ... */
     /* clear interrupt flags */
     ioctl(LVI, LVI_CLEAR_LOW_VOLTAGE_INT, LVI_27V_LEVEL);
}
else
{
     /* supply voltage is OK (above 2.7 V) */
}
```

This code tests if the supply voltage dropped under 2.7 V any time after it was previously tested.

5.5.3.43 LVI_CLEAR_LOW_VOLTAGE_INT - clears low voltage interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, LVI_CLEAR_LOW_VOLTAGE_INT,
           UWord16 param);
```

Arguments:

Table 5-142. LVI_CLEAR_LOW_VOLTAGE_INT ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the desired low voltage interrupt flag to be cleared. Use one of the predefined constants or a combination of more constants: LVI_INT | LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_CLEAR_LOW_VOLTAGE_INT* ioctl command clears low the voltage interrupt flags. Use predefined constants to define which flag to be cleared - LVI_22V_LEVEL for 2.2V interrupt, LVI_27V_LEVEL for 2.7V and LVI_INT for the combination of LVIS27 and LVIE27, or LVIS22 and LVIE22. This command clears directly the Power Supervisor Status Register bits - LVI (Bit 4), LVIS27S (Bit 3) and LVIS22S (Bit 2).

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *LVI_CLEAR_LOW_VOLTAGE_INT* command is implemented as a macro.

Example 5-128. LVI_CLEAR_LOW_VOLTAGE_INT

```
ioctl(LVI, LVI_CLEAR_LOW_VOLTAGE_INT, LVI_22V_LEVEL |
                                       LVI_27V_LEVEL);
```

This code clears both low voltage interrupt flags. This command can be used for example in the LVI interrupt service routine.

5.5.3.44 LVI_INT_ENABLE - enables low voltage interrupts

Call(s):

```
void ioctl(const int *pModuleBase, LVI_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-143. LVI_INT_ENABLE ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the desired low voltage interrupt flag to be enabled. Use one of the predefined constants or a combination of more constants: LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_INT_ENABLE* ioctl command enables low voltage interrupts. Use predefined constants to define which interrupt is to be enabled - LVI_22V_LEVEL for 2.2V interrupt, LVI_27V_LEVEL for 2.7V interrupt. This command writes directly to the Power Supervisor Control Register bits - LVIE27 (Bit 1) and LVIE22 (Bit 0).

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *LVI_INT_ENABLE* command is implemented as a macro.

Example 5-129. LVI_INT_ENABLE

```
ioctl(LVI, LVI_INT_ENABLE, LVI_22V_LEVEL | LVI_27V_LEVEL);
```

This code enables low voltage interrupts for both 2.2 V and 2.7 V levels.

5.5.3.45 LVI_INT_DISABLE - disables low voltage interrupts

Call(s):

```
void ioctl(const int *pModuleBase, LVI_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-144. LVI_INT_DISABLE ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the desired low voltage interrupt flag to be disabled. Use one of the predefined constants or a combination of more constants: LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_INT_DISABLE* ioctl command disables low voltage interrupts. Use predefined constants to define which interrupt is to be disabled - LVI_22V_LEVEL for 2.2V interrupt, LVI_27V_LEVEL for 2.7V. This command writes directly to the Power Supervisor Control Register bits - LVIE27 (Bit 1) and LVIE22 (Bit 0).

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *LVI_INT_DISABLE* command is implemented as a macro.

Example 5-130. LVI_INT_DISABLE

```
ioctl(LVI, LVI_INT_DISABLE, LVI_22V_LEVEL | LVI_27V_LEVEL);
```

This code disables low voltage interrupts for both voltage levels.

5.5.3.46 LVI_INT_SELECT - selects low voltage interrupts

Call(s):

```
void ioctl(const int *pModuleBase, LVI_INT_SELECT,
           UWord16 param);
```

Arguments:

Table 5-145. LVI_INT_SELECT ioctl call arguments

*pModuleBase	in	The Power Supervisor module identifier. Use LVI.
param	in	Parameter to select the desired low voltage interrupt flag to be enabled. Use one of the predefined constants or a combination of more constants: LVI_22V_LEVEL | LVI_27V_LEVEL

Description: The *LVI_INT_SELECT* ioctl command selects which low voltage interrupts are to be enabled. After this command is issued, the specified interrupts are enabled while the unspecified are disabled. Use predefined constants to define which only interrupt is to be enabled - LVI_22V_LEVEL for 2.2V interrupt, LVI_27V_LEVEL for 2.7V interrupt. This command writes directly to the Power Supervisor Control Register bits - LVIE27 (Bit 1) and LVIE22 (Bit 0).

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *LVI_INT_SELECT* command is implemented as a macro.

Example 5-131. LVI_INT_SELECT

```
ioctl(LVI, LVI_INT_SELECT, LVI_27V_LEVEL);
```

This code enables the low voltage interrupt for 2.7V level and disables the low voltage interrupt for 2.2V level.

5.5.3.47 SEMI_SET_DRIVE_BUS - controls external bus state when not accessed

Call(s):

```
void ioctl(const int *pModuleBase, SEMI_SET_DRIVE_BUS,
           UWord16 param);
```

Arguments:

Table 5-146. SEMI_SET_DRIVE_BUS ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.
param	in	Use one of predefined constants: SEMI_DRIVEN - external bus is in active state even when not accessed SEMI_NON_DRIVEN - external bus is not driven when not accessed (i.e. in tri-state or with pull-ups)

Description: The *SEMI_SET_DRIVE_BUS* ioctl command controls the state of the external bus when not accessed. Use SEMI_DRIVEN to leave the bus in active state all the time or SEMI_NON_DRIVEN to drive it only when accessed. This command writes directly in the Bus Control Register DRV ("Drive") bit (Bit 15).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_SET_DRIVE_BUS* command is implemented as a macro.

Example 5-132. SEMI_SET_DRIVE_BUS

```
ioctl(SEMI, SEMI_SET_DRIVE_BUS, SEMI_DRIVEN);
```

This code sets the external bus to be driven even when not accessed.

5.5.3.48 SEMI_WRITE_BASEREG*n* - writes SEMI Base Register

Call(s):

```
void ioctl(const int *pModuleBase, SEMI_WRITE_BASEREGn,
           UWord16 param);

      ... where n = 0..7
```

Arguments:

Table 5-147. SEMI_WRITE_BASEREGn ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.
param	in	Value to be written to the register

Description: The *SEMI_WRITE_BASEREGn* ioctl command writes specified value to one of the eight SEMI Base Registers. In the most cases the static SEMI configuration using the values from *appconfig.h* file is sufficient and the direct access to the SEMI registers is not needed.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_WRITE_BASEREGn* command is implemented as a macro.

Example 5-133. SEMI_WRITE_BASEREGn

```
ioctl(SEMI, SEMI_WRITE_BASEREG3, 0x100 | SEMI_CSBAR_BLKSZ_64K);
```

This code selects the Chip Select #3 to be active for the 64K memory block starting at address 0x10000.

5.5.3.49 SEMI_WRITE_OPTIONREG*n* - writes SEMI Option Register

Call(s):

```
void ioctl(const int *pModuleBase, SEMI_WRITE_OPTIONREGn,
           UWord16 param);

      ... where n = 0..7
```

Arguments:

Table 5-148. SEMI_WRITE_OPTIONREGn ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.
param	in	Value to be written to the register

Description: The *SEMI_WRITE_OPTIONREGn* ioctl command writes specified value to one of the eight SEMI Option Registers. In the most cases the static SEMI configuration using the values from *appconfig.h* file is sufficient and the direct access to the SEMI registers is not needed.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_WRITE_OPTIONREGn* command is implemented as a macro.

Example 5-134. SEMI_WRITE_OPTIONREGn

```
ioctl(SEMI, SEMI_WRITE_OPTIONREG3,
      SEMI_CSOR_BYTEEN_BOTH | SEMI_CSOR_RW_RO |
      SEMI_CSOR_PSDS_PSONLY | 0xA);
```

This code selects the memory bank identified by the Chip Select #3 to operate as 16bit read-only program memory with 10 wait states.

5.5.3.50 SEMI_WRITE_CONTROLREG - writes SEMI Bus Control Register

Call(s):

```
void ioctl(const int *pModuleBase, SEMI_WRITE_CONTROLREG,
           UWord16 param);
```

Arguments:

Table 5-149. SEMI_WRITE_CONTROLREG ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.
param	in	Value to be written to the register

Description: The *SEMI_WRITE_CONTROLREG* ioctl command writes specified value directly to the SEMI Bus Control Register. In the most cases the static SEMI configuration using the values from *appconfig.h* file is sufficient and the direct access to the SEMI registers is not needed.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_WRITE_CONTROLREG* command is implemented as a macro.

Example 5-135. SEMI_WRITE_CONTROLREG

```
ioctl(SEMI, SEMI_WRITE_CONTROLREG, 0x8000);
```

This code writes sets the DRV bit of the SEMI Bus Control Register.

5.5.3.51 SEMI_READ_BASEREG*n* - reads SEMI Base Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SEMI_READ_BASEREGn, NULL);

    ... where n = 0..7
```

Arguments:

Table 5-150. SEMI_READ_BASEREGn ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.

Description: The *SEMI_READ_BASEREGn* ioctl command reads the value of one of the eight SEMI Base Registers.

Returns: content of the selected SEMI Base Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_READ_BASEREGn* command is implemented as a macro.

Example 5-136. SEMI_READ_BASEREGn

```
UWord16 br3 = ioctl(SEMI, SEMI_READ_BASEREG3, NULL);
```

This code reads the SEMI Base Register for memory bank #3.

5.5.3.52 SEMI_READ_OPTIONREG*n* - reads SEMI Option Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SEMI_READ_OPTIONREGn,
          NULL);

      ... where n = 0..7
```

Arguments:

Table 5-151. SEMI_READ_OPTIONREGn ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.

Description: The *SEMI_READ_OPTIONREGn* ioctl command reads the value of one of the eight SEMI Option Registers.

Returns: content of the selected SEMI Option Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_READ_OPTIONREGn* command is implemented as a macro.

Example 5-137. SEMI_READ_OPTIONREGn

```
UWord16 or3 = ioctl(SEMI, SEMI_READ_OPTIONREG3, NULL);
```

This code reads the SEMI Option Register for memory bank #3.

5.5.3.53 SEMI_READ_CONTROLREG - reads SEMI Bus Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SEMI_READ_CONTROLREG,
          NULL);
```

Arguments:

Table 5-152. SEMI_READ_CONTROLREG ioctl call arguments

*pModuleBase	in	The External Memory Interface module identifier. Use SEMI. Note that SEMI is not available on all chips.

Description: The *SEMI_READ_CONTROLREG* ioctl command reads the value of the SEMI Bus Control Register.

Returns: content of the SEMI Bus Control Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SEMI_READ_CONTROLREG* command is implemented as a macro.

Example 5-138. SEMI_READ_CONTROLREG

```
UWord16 bcr = ioctl(SEMI, SEMI_READ_CONTROLREG, 0x8000);
```

This code reads the SEMI Bus Control Register.

5.5.4 SYS Driver Application

The SYS driver application is designed for Freescale/Motorola EVM's and intended to illustrate the usage of this driver by a real example. It shows a most common way of using the SYS/LVI module in user application.

The SYS driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\sys_demo* directory and consists of the application project *sys.mcp* and the source code for the application *main.c*. This demo application shows also the usage of the SYS and COP Drivers (Section 5.4).

Example 5-139. SYS Driver Application - appconfig.h

```
/*******************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
********************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*******************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.*****************************************************************************
*
*  File generated by Graphical Configuration Tool Fri, 09/Feb/2007, 12:34:57
*
*******************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Enable
    COP timeout: 8 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D
#define COP_COPCTL_INIT                  0x0002
#define COP_COPTO_INIT                   0xF423

/*.
```

```
    SYS Configuration
----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Enable
                      Low voltage 2.7V: Enable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
.*/
#define LVI_CONTROL_INIT                0x0003
#define SIM_GPS_INIT                    0x0000

/*.
    SEMI Configuration
----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                0x0005
#define SEMI_CSBAR1_INIT                0x0005
#define SEMI_CSBAR2_INIT                0x0005
#define SEMI_CSOR0_INIT                 0x1FC3
#define SEMI_CSOR1_INIT                 0x1BA3
#define SEMI_CSOR2_INIT                 0x1DA3

/*.
    INTC Configuration
----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
```

```
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0000
#define INT_VECTOR_ADDR_20              LowVoltageISR
#define INT_PRIORITY_LEVEL_20           INTC_LEVEL1

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: ISA0 , PullUp: Enable ,
    Pin  9: Function: ISA1 , PullUp: Enable ,
    Pin 10: Function: ISA2 , PullUp: Enable ,
.*/
#define GPIO_C_DDR_INIT                 0x000F
#define GPIO_C_PER_INIT                 0x07F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  7: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_DDR_INIT                 0x00C0
#define GPIO_D_PER_INIT                 0x1F01


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-140. SYS Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Demonstration application how to use SYS module
*
*              LEDS indicating the source of the last RESET:
*                RED LED    -> Power-ON RESET
*                YELLOW LED -> External RESET
*                GREEN LED  -> COP RESET (happens 8 sec after init)
*
*               Otehr LEDS indicating the state of application:
*                GREEN LED  -> flashing while in the main loop
*                YELLOW LED -> execution stuck in the LVI ISR
*
*
* TARGET: MC56F8xxx devices
*
*****************************************************************************/

#include "qs.h"

#include "sys.h"
#include "occs.h"
#include "intc.h"
#include "gpio.h"
#include "cop.h"

/* board-specific LEDs */
#include "../board.h"

/* forward prototypes */
void device_init(void);


/*
 *  The main
 */

void main (void)
{
    UWord16 i;

    /* initialize SYS, COP and pins */
    device_init();

    /* LEDs signal the source of last RESET */
    ioctl(GPIO_LEDS, GPIO_CLEAR_PIN, LED_Y | LED_G | LED_R);
    if (ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_POWER_ON_RESET))  /* PowerON RESET ->
RED LED */
        ioctl(GPIO_LEDS, GPIO_SET_PIN, LED_R);
    if (ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_EXTERN_RESET))    /* External RESET ->
YELLOW LED */
        ioctl(GPIO_LEDS, GPIO_SET_PIN, LED_Y);
    if (ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_COP_RESET))       /* COP RESET -> GREEN
LED */
        ioctl(GPIO_LEDS, GPIO_SET_PIN, LED_G);
```

```
    /* clear reset status */
    ioctl(SYS, SYS_CLEAR_RESET_SOURCE, SYS_ALL_RESETS);

    /* low voltage flags can be set after power up in some situations, clear them */
    ioctl(LVI, LVI_CLEAR_LOW_VOLTAGE_INT, LVI_22V_LEVEL | LVI_27V_LEVEL);

    /* configure Interrupt Controller (IPR) */
    ioctl(INTC, INTC_INIT, NULL);

    /* enable interrupts in SR */
    archEnableInt();

    /* main program loop */
    while (1)
    {
        /* Toggle 2nd GREEN LED to show an application is running */
        ioctl(GPIO_LED_G2, GPIO_TOGGLE_PIN, LED_G2);

        for (i=0; i<100; i++)
            archDelay(0xffff);

        /* NOTE: we do let COP watchdog to expire */
    }
}

/*************************************************************************
    interrupt subroutine for low voltage detection (brown-out)
*************************************************************************/

#pragma interrupt on

void LowVoltageISR(void)
{
    /* supply voltage dropped below threshold */

    /* insert here a code to:
         - switch off power outputs and put them to safe state
           to preserve system failure during Vcc fall
         - put peripherals to safe state (especially PWM etc.)
    */

    /* 2nd YELLOW LED on */
    ioctl(GPIO_LED_Y2, GPIO_SET_PIN, LED_Y2);

    /* switch processor clock to prescaler clock (bypass the PLL) */
    /* to maintain operation from 1.8V and up */
    ioctl(OCCS, OCCS_SET_ZCLOCK_SOURCE, OCCS_PRESCALER_OUTPUT);

    /* for 8MHz XTAL - processor is now running on 8MHz, IPbus 4MHz (peripherals) */

    /* wait untill voltage rises again */
    while (ioctl(LVI, LVI_GET_NONSTICKY_INT_SOURCE, LVI_22V_LEVEL | LVI_27V_LEVEL))
        ;

    /* clear combined low voltage interrupt flag, the two sticky sources may remain
asserted */
    ioctl(LVI, LVI_CLEAR_LOW_VOLTAGE_INT, LVI_INT);

    /* Vcc rose up to sufficient level again, normal operation will continue */
    /* if Vcc drops below Power-up level (1.8V) then recover will occur by RESET */
    ioctl(OCCS, OCCS_SET_ZCLOCK_SOURCE, OCCS_POSTSCALER_OUTPUT);

    /* power outputs should be put to previous state */
    /* and also peripherals should be set to state for normal operation */
```

```
    /* turn off YELLOW LED again */
    ioctl(GPIO_LED_Y2, GPIO_CLEAR_PIN, LED_Y2);
}

#pragma interrupt off



/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.6 PMC Driver

This section describes the API of MC56F800x Power Management Controller (PMC) on-chip module. The functionality of the PMC module itself is described in the **56F800x Peripheral Reference Manual**.

5.6.1 Introduction

The MC56F800x devices have the Power Management Controller module. In this module is integrated Power-On Reset, Out-Of-Regulation detection with interrupt capability and Low-Voltage Detectwith reset or interrupt capability functions. This module controls a buffer for a bandgap reference voltage output and programable LVD trip points. There is also possibility to set power saving modes.

This section describes the PMC driver software, providing the low level software layer interfacing hardware with software.

5.6.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.6.3.

Table 5-153. PMC Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
PMC (PMC_BASE)	0xF260	N/A	N/A	N/A

5.6.2.1 API Definition

The following header files are needed in order to use the PMC device driver:

Required Header File(s):

```
#include "qs.h"
#include "pmc.h"
```

The following information may be found in the header file *pmc.h*.

Public Data Structure(s):

None

5.6.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static PMC module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-154. PMC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
PMC_SCR_INIT	UWord16	The initial value of the PMC Status and Control Register.
PMC_CR2_INIT	UWord16	Initial value of the PMC Control Register.

5.6.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the PMC device modes or accesses the PMC register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-155. PMC Driver Arguments - ioctl

pModuleBase	in	PMC module identifier. Use PMC.
cmd	in	Commands found in *pmc.h* which are used to modify the PMC module status and control registers. See Table 5-156.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-156. ioctl commands

cmd	param	Return	Description
PMC_INIT	NULL	None	Initializes PMC module by data from the configuration file (appconfig.h).
PMC_CLEAR_FLAGS	PMC_FLAG_OUT_REG | PMC_FLAG_LOW_VOLT | PMC_FLAG_RESET | PMC_FLAG_ALL | PMC_FLAG_PART_ POWER_DOWN	None	Clears selected interrupt and status flags.
PMC_TEST_FLAGS	PMC_FLAG_OUT_REG | PMC_FLAG_LOW_VOLT | PMC_FLAG_RESET | PMC_FLAG_PART_ POWER_DOWN	None	Tests selected interrupt and status flags.
PMC_SET_INT_ENABLE	PMC_INT_OUT_REG | PMC_INT_LOW_VOLT	None	Enables selected interrupts.
PMC_SET_INT_DISABLE	PMC_INT_OUT_REG | PMC_INT_LOW_VOLT	None	Disables selected interrupts.
PMC_SET_LOW_VOLTAGE_ RESET	PMC_ENABLE / PMC_DISABLE	None	Enables or disables hardware reset when LVDF=1.
PMC_SET_PARTIAL_ POWER_DOWN	PMC_ENABLE / PMC_DISABLE	None	Enables or disables partial power down mode.
PMC_SET_LOW_POWER_ REGULATOR_WAIT_MODES	PMC_ENABLE / PMC_DISABLE	None	Enables or disables power wait modes.
PMC_TEST_LOW_POWER_ REGULATOR_STATUS	NULL	None	Tests low power regulator status.
PMC_SET_LOW_POWER_ WAKEUP_INTERRUPT	PMC_ENABLE / PMC_DISABLE	None	Enables or disables low power wakeup when an interrupt is occured.
PMC_SET_BANDGAP_ BUFFER	PMC_ENABLE / PMC_DISABLE	None	Enables or disables an internal buffer for bandgap bandgap voltage reference.
PMC_SET_LOW_VOLTAGE_ DETECTOR_ENABLE	NULL	None	Enables low voltage detector.
PMC_SET_LOW_VOLTAGE_ DETECTOR_DISABLE	NULL	None	Disables low voltage detector.

Table 5-156. ioctl commands (Continued)

cmd	param	Return	Description
PMC_SET_LOW_VOLTAGE_ DETECTOR_LEVEL	PMC_LOW_VOLT_SEL_ LVDL / PMC_LOW_VOLT_SEL_ LVDH/	None	Selects low voltage detector level.
PMC_SET_WPROTECTION	PMC_ENABLE / PMC_DISABLE / PMC_ENABLE_PERMAN ENT / PMC_DISABLE_PERMAN ENT	None	Write-protects PMC configuration and TRIM values.
PMC_SET_1KHZ_OSC	PMC_ENABLE / PMC_DISABLE	None	Enables or disables an internal low power 1kHz oscillator.
PMC_SET_1KHZ_OSC_TRIM	PMC_1KHZ_TRIM_p24_7 5 / PMC_1KHZ_TRIM_p16_5/ PMC_1KHZ_TRIM_p8_25/ PMC_1KHZ_TRIM_CENT ER / PMC_1KHZ_TRIM_n8_25/ PMC_1KHZ_TRIM_n16_5/ PMC_1KHZ_TRIM_n24_7 5 / PMC_1KHZ_TRIM_n33/	None	Sets an internal low power 1kHz oscillator trim value.
PMC_SET_1KHZ_OSC_ FACTORY_TRIM	NULL	None	Inserts to the 1kHz Oscillator Trim Bits the factory trim value which is stored in procesor memory.
PMC_SET_LVD_TRIM	PMC_LVD_TRIM_p14 / PMC_LVD_TRIM_p13 / PMC_LVD_TRIM_p12 / ... PMC_LVD_TRIM_n5 / PMC_LVD_TRIM_n6 / PMC_LVD_TRIM_ CENTER Note that list of all parame-ters see *pmc.h*.	None	Sets the Low-Voltage Detector Trim value.
PMC_SET_LVD__FACTORY_ TRIM	NULL	None	Inserts to the Low Voltage Detector Ttim Bits the factory trimvalue which is stored in procesor mem-ory.

5.6.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.6.3.1 PMC_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, PMC_INIT, NULL);
```

Arguments:

Table 5-157. PMC_INIT ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_INIT* ioctl command executes the initialization function, which initializes all configured PMC registers by the values defined in *appconfig.h*. It is intended for static configuration of the PMC module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_INIT* ioctl command is implemented as a function call. Only necessary PMC registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-141. PMC_INIT

```
ioctl(PMC, PMC_INIT, NULL);
```

This code initializes the PMC modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.6.3.2 PMC_CLEAR_FLAGS - clear interrupt and status flags

Call(s):

```
void ioctl(const int *pModuleBase, PMC_CLEAR_FLAGS,
           UWord16 param);
```

Arguments:

Table 5-158. PMC_CLEAR_FLAGS ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use combination of the predefined constants: PMC_FLAG_OUT_REG | ... to clear the Out of Regulation interrupt flag. PMC_FLAG_LOW_VOLT | ... to clear the Low Voltage Detect interrupt flag. PMC_FLAG_RESET | ... to clear the Power On Reset flag. PMC_FLAG_PART_POWER_DOWN | ... to clear the Partial Power Down flag. PMC_FLAG_ALL ... to clear all flags.

Description: The *PMC_CLEAR_FLAGS* ioctl command clears the selected interrupt flags. This command clears the Out of Regulation interrupt flag, when PMC_FLAG_OUT_REG is used as parameter. With the PMC_FLAG_LOW_VOLT parameter this commaned clears the Low Voltage Detect interrupt flag. With the PMC_FLAG_RESET parameter this commaned clears the Power On Reset status flag and with the PMC_FLAG_PART_POWER_DOWN parameter clears the Partial Power Down flag.

This command clears flags writig one to the Out of Regulation Flag (OORF) bit, to the Low Voltage Detect Flag (LVDF), to the Partial Power Down Flag (PPDF) and to the Power On Reset Flag (PORF) of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_CLEAR_FLAGS* command is implemented as a macro.

Example 5-142. PMC_CLEAR_FLAGS

```
ioctl(PMC, PMC_CLEAR_FLAGS, PMC_FLAG_LOW_VOLT | PMC_FLAG_RESET);
```

This code clears the Low Voltage interrupt flag and the Power On Reset status flag.

5.6.3.3 PMC_TEST_FLAGS - test interrupt and status flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PMC_TEST_FLAGS,
              UWord16 param);
```

Arguments:

Table 5-159. PMC_TEST_FLAGS ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use combination of the predefined constants: PMC_FLAG_OUT_REG | ... to test the Out of Regulation interrupt flag. PMC_FLAG_LOW_VOLT |... to test the Low Voltage Detect interrupt flag. PMC_FLAG_RESET | ... to test the Power On Reset flag. PMC_FLAG_PART_POWER_DOWN ... to test the Partial Power Down flag.

Description: The *PMC_TEST_FLAGS* ioctl command returns state of selected interrupt flags. This command returns value of the Out of Regulation interrupt flag, when PMC_FLAG_OUT_REG is used as parameter. With the PMC_FLAG_LOW_VOLT parameter this commaned returns value of the Low Voltage Detect interrupt flag. With the PMC_FLAG_RESET parameter this commaned returns value of the Power On Reset status flag and with the PMC_FLAG_PART_POWER_DOWN parameter returns value of the Partial Power Down flag.

This command reads and returns values of the Out of Regulation Flag (OORF) bit, the Low Voltage Detect Flag (LVDF), the Partial Power Down Flag (PPDF) and the Power On Reset Flag (PORF) of the PMC Status and Control Register.

Returns: UWord16, Non-zero value when at least one of specified bits is set. Zero when all of the specified bits are clear.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_TEST_FLAGS* command is implemented as a macro.

Example 5-143. PMC_TEST_FLAGS

```
ioctl(PMC, PMC_TEST_FLAGS, PMC_FLAG_LOW_VOLT | PMC_FLAG_RESET);
```

This code clears the Low Voltage interrupt flag and the Power On Reset status flag.

5.6.3.4 PMC_SET_INT_ENABLE - enable PMC Interrupts

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-160. PMC_SET_INT_ENABLE ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use the combination of interrupt control bits PMC_INT_OUT_REG | ... to enable the Out of Regulation interrupt PMC_INT_LOW_VOLT... to enable theLow Voltage Detect interrupt

Description: The *PMC_SET_INT_ENABLE* ioctl command enables selected PMC interrupts. The *PMC_SET_INT_ENABLE* ioctl command modifies (sets) the appropriate bits in the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PMC_SET_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-144. PMC_SET_INT_ENABLE

```
ioctl(PMC, PMC_SET_INT_ENABLE, PMC_INT_LOW_VOLT);
```

This code enables the Low Voltage Detect interrupt.

5.6.3.5 PMC_SET_INT_DISABLE - disable PMC Interrupts

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-161. PMC_SET_INT_DISABLE ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use the combination of interrupt control bits PMC_INT_OUT_REG | ... to disable the Out of Regulation interrupt PMC_INT_LOW_VOLT... to disable the Low Voltage Detect interrupt

Description: The *PMC_SET_INT_DISABLE* ioctl command disables selected PMC interrupts. The *PMC_SET_INT_DISABLE* ioctl command modifies (clears) the appropriate bits in the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x..

Design/Implementation: The *PMC_SET_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-145. PMC_SET_INT_DISABLE

```
ioctl(PMC, PMC_SET_INT_DISABLE, PMC_INT_OUT_REG);
```

This code disables the Out of Regulation interrupt.

5.6.3.6 PMC_SET_LOW_VOLTAGE_RESET - enable/disable Low Voltage Hardware reset

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_LOW_VOLTAGE_RESET,
     UWord16 param);
```

Arguments:

Table 5-162. PMC_SET_LOW_VOLTAGE_RESET ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENABLE ... to enable the Low Voltage Hardware reset PMC_DISABLE ... to disable the Low Voltage Hardware reset

Description: The *PMC_SET_LOW_VOLTAGE_RESET* ioctl command enables or disables the hardware processor reset when when low voltage detector set up LVDF bit to 1.

This command writes into the Low Voltage Detect Reset Enable (LVDRE) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LOW_VOLTAGE_RESET* command is implemented as a macro.

Example 5-146. PMC_SET_LOW_VOLTAGE_RESET

```
ioctl(PMC, PMC_SET_LOW_VOLTAGE_RESET, PMC_ENABLE)
```

This code enables the Low Voltage Hardware Reset.

5.6.3.7 PMC_SET_PARTIAL_POWER_DOWN - enable/disable Partial Power-Down mode

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_PARTIAL_POWER_DOWN,
     UWord16 param);
```

Arguments:

Table 5-163. PMC_SET_PARTIAL_POWER_DOWN ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENABLE ... to enable the Partial Power-Down mode PMC_DISABLE ... to disable the Partial Power Down mode

Description: The *PMC_SET_PARTIAL_POWER_DOWN* ioctl command enables or disables the Partial Power-Down mode. This command configures Power Management Controller to enter partial power-down mode the next time that the STOP command is executed.

This command writes into the Partial Power-Down Enable (PPDE) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_PARTIAL_POWER_DOWN* command is implemented as a macro.

Example 5-147. PMC_SET_PARTIAL_POWER_DOWN

```
ioctl(PMC, PMC_SET_PARTIAL_POWER_DOWN, PMC_DISABLE)
```

This code disables to enter the DSP controller into the Partial Power Down mode.

5.6.3.8 PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES - enable/disable power wait modes

Call(s):

```
void ioctl(const int *pModuleBase,
     PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES,
     UWord16 param);
```

Arguments:

Table 5-164. PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENTER ... to request the low-power run and low-power wait modes PMC_DISABLE ... to disable the low-power run and wait modes

Description: The *PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES* ioctl command controls entry into the low-power run and low-power wait modes in which the voltage regulator is put into standby.

This command writes into the Low Power Regulator Control (LPR) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES* command is implemented as a macro.

Example 5-148. PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES

```
ioctl(PMC, PMC_SET_LOW_POWER_REGULATOR_WAIT_MODES, PMC_ENTER)
```

This code requests the low-power run and the low-power wait modes.

5.6.3.9 PMC_TEST_LOW_POWER_REGULATOR_STATUS - read regulator status

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
      PMC_TEST_LOW_POWER_REGULATOR_STATUS,
      NULL);
```

Arguments:

Table 5-165. PMC_TEST_LOW_POWER_REGULATOR_STATUS ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_TEST_LOW_POWER_REGULATOR_STATUS* ioctl command reads and returns state of the Low-Power Regulator Status (LPRS) bit of the PMC Status and Control Register.

Returns: UWord16, Non zero value, when the voltage regulator has entered into standby for the low-power run or wait mode.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_TEST_LOW_POWER_REGULATOR_STATUS* command is implemented as a macro.

Example 5-149. PMC_TEST_LOW_POWER_REGULATOR_STATUS

```
if (ioctl(PMC, PMC_TEST_LOW_POWER_REGULATOR_STATUS, NULL))
{
      ...
}
```

This code tests, if the voltage regulator entered into standby for the low-power run or low-power wait mode.

5.6.3.10 PMC_SET_LOW_POWER_WAKEUP_INTERRUPT- enable/disable low power wakeup on an interrupt

Call(s):

```
void ioctl(const int *pModuleBase,
      PMC_SET_LOW_POWER_WAKEUP_INTERRUPT,
      UWord16 param);
```

Arguments:

Table 5-166. PMC_SET_LOW_POWER_WAKEUP_INTERRUPT ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENABLE PMC_DISABLE

Description: The *PMC_SET_LOW_POWER_WAKEUP_INTERRUPT* ioctl command enables or disables the low-power wakeup on interrupt. This command with the PMC_ENABLE parameter activates the volatage regulator exits standby when any active MCU interrupt occured.

This command writes into the Low-Power Wakeup on Interrupt (LPWUI) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LOW_POWER_WAKEUP_INTERRUPT* command is implemented as a macro.

Example 5-150. PMC_SET_LOW_POWER_WAKEUP_INTERRUPT

```
ioctl(PMC, PMC_SET_LOW_POWER_WAKEUP_INTERRUPT, PMC_DISABLE)
```

This code disables the voltage regulator exits standby when any activate MCU interrupt occures.

5.6.3.11 PMC_SET_BANDGAP_BUFFER- enable/disable bandgap buffer

Call(s):

```
void ioctl(const int *pModuleBase,PMC_SET_BANDGAP_BUFFER,
     UWord16 param);
```

Arguments:

Table 5-167. PMC_SET_BANDGAP_BUFFER ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENABLE ... to enable the buffer for the bandgap voltage reference PMC_DISABLE ... to disable the buffer for the bandgap voltage reference

Description: The *PMC_SET_BANDGAP_BUFFER* ioctl command enables or disables the buffer for the bandgap voltage reference. The bandgap voltage reference is used by the ADC module and onchip comparators.

This command writes into the Bandgap Buffer Enable (BGBE) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_BANDGAP_BUFFER* command is implemented as a macro.

Example 5-151. PMC_SET_BANDGAP_BUFFER

```
ioctl(PMC, PMC_SET_BANDGAP_BUFFER, PMC_ENABLE)
```

This code enables the buffer for the bandgap voltage reference.

5.6.3.12 PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE - enable low voltage detector

Call(s):

```
void ioctl(const int *pModuleBase,
           PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE, NULL);
```

Arguments:

Table 5-168. PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE* ioctl command enables the low voltage detector.

This command writes into the Low-Voltage Detect Enable (LVDE) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE* ioctl command is implemented as a macro.

Example 5-152. PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE

```
ioctl(PMC, PMC_SET_LOW_VOLTAGE_DETECTOR_ENABLE, NULL);
```

This code enables the Low-Voltage Detector.

5.6.3.13 PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE - disable low voltage detector

Call(s):

```
void ioctl(const int *pModuleBase,
           PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE, NULL);
```

Arguments:

Table 5-169. PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE* ioctl command disables the low voltage detector.

This command writes into the Low-Voltage Detect Enable (LVDE) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE* ioctl command is implemented as a macro.

Example 5-153. PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE

```
ioctl(PMC, PMC_SET_LOW_VOLTAGE_DETECTOR_DISABLE, NULL);
```

This code disables the Low-Voltage Detector.

5.6.3.14 PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL - select low-voltage detector level

Call(s):

```
void ioctl(const int *pModuleBase,
      PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL,
      UWord16 param);
```

Arguments:

Table 5-170. PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_LOW_VOLT_SEL_VLDL ... to select the VLDL level (1.86V) PMC_LOW_VOLT_SEL_VLDH ... to select the VLDH level l(2.33V)

Description: The *PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL* ioctl command selects the Low-Voltage detector trip point voltage level

This command writes into the Low-Voltage Detect Level Select (LVLS) bit of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL* command is implemented as a macro.

Example 5-154. PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL

```
ioctl(PMC, PMC_SET_LOW_VOLTAGE_DETECTOR_LEVEL,
      PMC_LOW_VOLT_SEL_VLDL)
```

This code selects the Low-Voltage detector trip point to the VLDL value (1.86V).

5.6.3.15 PMC_SET_WPROTECTION - protect power management controler settings

Call(s):

```
void ioctl(const int *pModuleBase,
     PMC_SET_WPROTECTION, UWord16 param);
```

Arguments:

Table 5-171. PMC_SET_WPROTECTION ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the following constants **PMC_ENABLE** ... enable protection (may be changed later) **PMC_ENABLE_PERMANENT** ... enable protection until reset **PMC_DISABLE** ... disable protection (may be re-enabled later) **PMC_DISABLE_PERMANENT** ... disable protection until reset

Description: The *PMC_SET_WPROTECTION* ioctl command write-protects the Power Management Controler configuration bits in the PMC module. Depending on the parameter value, the protection may be activated or deactivated permanently (until the next reset).

The PMC_SET_WPROTECTION command write-protects the SCR and CR2 registers in the PMC module.

This command writes directly to the PROT bit-field of the PMC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This commands are applicable only on MC56F800x..

Design/Implementation: The PMC_SET_WPROTECTION command is implemented as a macro.

Example 5-155. PMC_SET_WPROTECTION

```
ioctl(PMC, PMC_SET_WPROTECTION, SYS_DISABLE_PERMANENT);
```

This code disables write-protection of the PMC settings. It will not be possible to enable this protection until the next reset occurs.

5.6.3.16 PMC_SET_1KHZ_OSC - enable/disable 1kHz oscillator

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_1KHZ_OSC,
     UWord16 param);
```

Arguments:

Table 5-172. PMC_SET_1KHZ_OSC ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_ENABLE PMC_DISABLE

Description: The *PMC_SET_1KHZ_OSC* ioctl command enables or disables the 1kHz low power oscillator.

This command writes into the 1KHZ Oscillator Enable (LPO_EN) bit of the PMC Control Register.2

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_1KHZ_OSC* command is implemented as a macro.

Example 5-156. PMC_SET_1KHZ_OSC

```
ioctl(PMC, PMC_SET_1KHZ_OSC, PMC_DISABLE)
```

This code disables the 1kHz low power oscillator.

5.6.3.17 PMC_SET_1KHZ_OSC_TRIM - select 1kHz oscillator trim value

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_1KHZ_OSC_TRIM,
     UWord16 param);
```

Arguments:

Table 5-173. PMC_SET_1KHZ_OSC_TRIM ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_1KHZ_TRIM_p24_75 / PMC_1KHZ_TRIM_p16_5 / PMC_1KHZ_TRIM_p8_25 / PMC_1KHZ_TRIM_CENTER / PMC_1KHZ_TRIM_n8_25 / PMC_1KHZ_TRIM_n16_5 / PMC_1KHZ_TRIM_n24_75 / PMC_1KHZ_TRIM_n33

Description: The *PMC_SET_1KHZ_OSC_TRIM* ioctl command selects the 1kHz low power oscillator trim value. The frequency of the 1kHz low power oscillator can be adjusted in range from +24,75% to -33% in 7 steps. This command selects positive frequency adjust 24,75%, when PMC_1HKZ_TRIM_p24_75 is used as parameter. With the PMC_1HKZ_TRIM_n16_5 this command selects negative frequency adjust 16,5%.

This command writes into the 1KHZ Oscillator Trim (LPO_TRIM) bits of the PMC Control Register.2

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_1KHZ_OSC_TRIM* command is implemented as a macro.

Example 5-157. PMC_SET_1KHZ_OSC_TRIM

```
ioctl(PMC, PMC_SET_1KHZ_OSC_TRIM, PMC_1KHZ_TRIM_CENTER)
```

This code disables the 1kHz oscillator adjust settings.

5.6.3.18 PMC_SET_1KHZ_OSC_FACTORY_TRIM - Select 1kHz oscillator factory trim value

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_1KHZ_OSC_FACTORY_TRIM,
     NULL);
```

Arguments:

Table 5-174. PMC_SET_1KHZ_OSC_FACTORY_TRIM ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_SET_1KHZ_OSC_FACTORY_TRIM* ioctl command reads factory trim value from flash memory and writes into the 1KHZ Oscillator Trim (LPO_TRIM) bits of the PMC Control Register.2

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_1KHZ_OSC_FACTORY_TRIM* command is implemented as a macro.

Example 5-158. PMC_SET_1KHZ_OSC_FACTORY_TRIM

```
ioctl(PMC, PMC_SET_1KHZ_OSC_FACTORY_TRIM, NULL)
```

This code adjusts the raw frequency to 1kHz by the factory trim value stored in the processor flash memory.

5.6.3.19 PMC_SET_LVD_TRIM - select LVD trim value

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_LVD_TRIM,
     UWord16 param);
```

Arguments:

Table 5-175. PMC_SET_LVD_TRIM ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.
param	in	Use one of the predefined constants: PMC_LVD_TRIM_p14 / PMC_LVD_TRIM_p13 / PMC_LVD_TRIM_p12 / PMC_LVD_TRIM_n5 / PMC_LVD_TRIM_n6 / PMC_LVD_TRIM_CENTER Note that list of all parameters see *pmc.h*.

Description: The *PMC_SET_LVD_TRIM* ioctl command selects the regulator trim increment.

This command writes into the Regulator Trim (PMC_TRIM) bits of the PMC Control Register.2

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LVD_TRIM* command is implemented as a macro.

Example 5-159. PMC_SET_LVD_TRIM

```
ioctl(PMC, PMC_SET_LVD_TRIM, PMC_LVD_TRIM_p1)
```

This code sets trim increment to +1.

5.6.3.20 PMC_SET_LVD_FACTORY_TRIM - Select LVD factory trim value

Call(s):

```
void ioctl(const int *pModuleBase, PMC_SET_LVD_FACTORY_TRIM,
     NULL);
```

Arguments:

Table 5-176. PMC_SET_LVD_FACTORY_TRIM ioctl call arguments

*pModuleBase	in	The PMC module identifier. Use PMC.

Description: The *PMC_SET_LVD_FACTORY_TRIM* ioctl command reads factory trim increment value from flash memory and writes into the PMC Trim Increment (PMC_TRIM) bits of the PMC Control Register.2

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PMC_SET_LVD_FACTORY_TRIM* command is implemented as a macro.

Example 5-160. PMC_SET_LVD_FACTORY_TRIM

```
ioctl(PMC, PMC_SET_LVD_FACTORY_TRIM, NULL)
```

This code sets the Trim Increment by the factory trim value stored in the processor flash memory.

5.6.4 PMC Driver Applications

There is a sample applications demonstrating how PMC module can be used in a user application. The supported EVM board for PMC sample application is MC56F8006DEMO.

5.6.4.1 pmc_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\pmc_demo

This application shows how can be handled Low-Voltage detector interrupt and how can be status flags tested after power reset.LEDS indicating the source of the last RESET:

- RED LED -> Power-ON RESET
- YELLOW LED -> External RESET
- GREEN LED -> COP RESET (happens 8 sec after init)

Otehr LEDS indicating the state of application:

- GREEN LED -> flashing while in the main loop
- YELLOW LED -> execution stuck in the LVI ISR
- RED LED -> toggle led every execution LVI ISR

5.7 FlexCAN Driver

This section describes the API for the 56F800E FlexCAN module. The functionality of the FlexCAN module itself is described in the **MC56F8300 Peripheral User Manual**.

5.7.1 Introduction

The FlexCAN module is a communication controller implementing the CAN protocol. The module contains 16 Message Buffers (MBs) which can be assigned as either a Transmit (CANTx) buffers or a Receive (CANRx) buffers. Each MB has also assigned an interrupt flag bit, to indicate successful completion of transmission or reception, respectively.

The main FlexCAN module features are:

- Based on and includes all existing superset Motorola-Toucan module
- Full implementation of the CAN protocol specification - Version 2.0
  - — Standard Data and Remote Frames (up to 109 bits long)
  - — Extended Data and Remote Frames (up to 127 bits long)
  - — 0-8 bytes data length
  - — Programmable Bit Rate up to 1Mbit/sec
- Up to 16 flexible Message Buffers (MBs) of 0-8 bytes Data Length, each configurable as CANRx or CANTx, all support Standard and Extended Messages
- Listen-only mode capability
- Three programmable mask registers
  - — global (for MBs 0-13)
  - — special for MB14
  - — special for MB15
- Programmable transmit-first scheme: Lowest ID or lowest buffer number
- Time Stamp, based on 16-bit free-running timer
- Global network time, synchronized by a specific message
- Maskable interrupts
- Low power Sleep mode, with programmable Wake Up on bus activity

The following sections describe the FlexCAN driver software which provides the low level API to the FlexCAN hardware.

5.7.2 Quick Reference

This section defines the terms and formulas used later in Section 5.7.3.

Table 5-177. FlexCAN Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F83xx	MC56F836x
FlexCAN (CAN_BASE)	N/A	N/A (see msCAN)	0xF800	0xF800
FlexCAN2 (CAN2_BASE)	N/A	N/A	N/A	0xFA00

5.7.2.1 FlexCAN Bit-Timing

The FlexCAN module supports a variety of means to setup the bit timing required by the CAN protocol. The main FlexCAN clock is called serial clock or *sclock*. The ratio between system clock and *sclock* can be specified using the PRES_DIV prescaler value in the FCCTL1 register.

sclock = system clock / (FCCTL1.PRES_DIV+1)

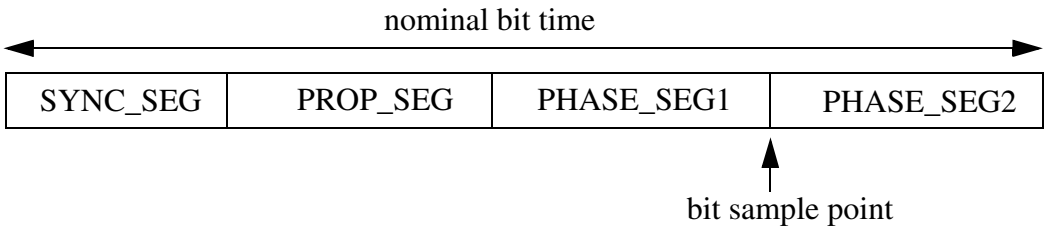
A single *sclock* cycle defines a basic time unit called “time quantum” (tq). All other FlexCAN timing parameters are measured in the time quanta units.

1/sclock = 1 time quantum

The nominal bit rate of the CAN bus is based on the nominal bit time interval.

nominal bit rate = 1 / nominal bit time

The nominal bit time is split to four non-overlapping time intervals, each measured in time quanta units.



SYNC_SEG - Synchronization Segment

This part of the bit time is used to synchronize the various nodes on the bus. An edge is expected to lie within this segment. This part is always 1 tq long.

SYNC_SEG = 1 [tq]

PROP_SEG - Propagation Segment

This part of the bit time is used to compensate for the physical delay times within the network. It is twice the sum of the signal's propagation time on the bus line, the input comparator delay, and the output driver delay. The PROPSEG field of the FCCTL0 contains the length of the PROP_SEG part in tq units

PROP_SEG = FCCTL0.PROPSEG + 1 [tq]

PHASE_SEG1, PHASE_SEG2 - Phase Buffer Segments

These Phase Buffer Segments are used to compensate the edge phase errors. These segments can be lengthened or shortened by re-synchronization (SJW parameter). The length of Phase Buffer Segments can be configured in PSEG1 and PSEG2 fields of the FCCTL1 register.

PHASE_SEG1 = FCCTL1.PSEG1 + 1 [tq]

PHASE_SEG2 = FCCTL1.PSEG2 + 1 [tq]

RE-SYNCHRONIZATION JUMP WIDTH (RJW)

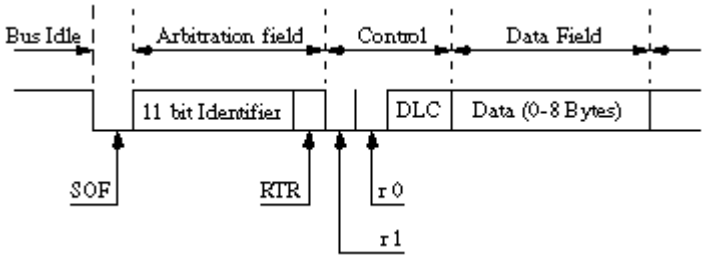
As a result of re-synchronization, PHASE_SEG1 may be lengthened or PHASE_SEG2 may be shortened. The amount of lengthening or shortening of the Phase Buffer Segments has an upper bound given by the Re-synchronization Jump Width. This parameter can be configured in RJW field of the FCCTL1 register.

RJW = FCCTL1.RJW + 1 [tq]

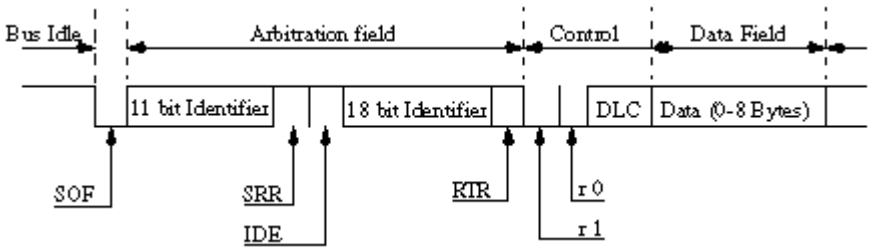
For more information about bit timing requirements, see the CAN 2AB standard document or Motorola Application Note number 1798 (order number AN1798/D).

5.7.2.2 FlexCAN Message Identifiers

CAN Standard 2.0 defines two kinds of CAN message identifiers. Part A of the standard defines “standard” 11 bit message identifier and part B defines “extended” 29 bit identifier. The CAN message header contains control bit named “ID Extended” (IDE) which, when set, identifies message with 29 bit extended identifier. In Part A-message format the IDE bit is reserved for future use (“r1”) and has always a value of zero.



The Remote Transmit Request (RTR) bit, which follows the 11 bit identifier in standard ID messages is replaced with “dummy” Substitute Remote Request (SRR) bit in extended 29 bit ID messages. The SRR is always transmitted as one. For extended ID messages, 18 ID bits added and a new RTR bit are appended after IDE bit.



There are several FlexCAN registers where the message ID is to be specified. The examples are Message Buffer ID registers, Global Mask Registers and Mask Registers for MB14 and MB15. Various registers containing the message ID have always the same format, which results from the physical bit sequence as seen on the bus. High part of the ID register (ID_HIGH) contains bits ID28 to ID18 - which store the all 11 bits of the standard identifier or the upper 11 bits of the extended identifier. ID_HIGH also contains RTR for standard identifier (SRR=1 for extended id) and IDE bit identifying the type of the identifier. Lower part of the ID register (ID_LOW) contains the rest of the ID bits and RTR for the extended identifier.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ID_HIGH	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21	ID20	ID19	ID18	RTR SRR	IDE	ID17	ID16	ID15

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID_LOW	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

The FlexCAN low-level driver of DSP56800E_Quick_Start contains two inline methods to convert numerical representation of message ID from and to the 32-bit value, which is suitable for writing into ID_HIGH and ID_LOW registers.

```
inline UWord32 FCAN_Id2Idr(UWord32 id);
```

This inline function converts the numerical representation of the ID value into the “raw” format suitable for FlexCAN registers. The most significant bit (bit31) of the ID value is used to identify extended message identifier. The valid range of the input parameter is 0x000...0x7ff (standard identifier) and 0x80000000...0x9fffffff (extended identifier). The symbol FCAN_ID_EXT is defined in *fcan.h* as the MSB (0x80000000), so it can be used OR-ed with the numerical ID representation when specifying extended identifiers.

```
inline UWord32 FCAN_Idr2Id(UWord32 idr);
```

This inline function parses the “raw” value read from two FlexCAN ID registers into numerical representation. MSB of the returned value is set for extended identifiers.

The FlexCAN ioctl commands accept and return the numerical representation of the ID values (plus the MSB to identify extended identifiers) and use the conversion functions described above implicitly (e.g. *FCAN_SET_RXGMASK* or *FCANMB_SET_ID* commands[1]). The ioctl commands

1. An exception is the FCANMB_GET_ID command, which is implemented as a optimized assembly function in *fcan.c*.

operating directly with the “raw” ID value (sometimes referred as IDR value) without further conversions have the *_RAW* suffix (e.g. *FCAN_SET_RXGMASK_RAW* or *FCANMB_GET_ID_RAW*).

5.7.2.3 API Definition

The following header files are needed in order to use the FlexCAN device driver:

Required Header File(s):

```
#include "qs.h"
#include "fcan.h"
```

The following information may be found in the header file *fcan.h*.

Public Data Structure(s):

```
none
```

General-Use Macros and inline functions:

```
inline UWord32 FCAN_Id2Idr(UWord32 id);
inline UWord32 FCAN_Idr2Id(UWord32 idr);
```

5.7.2.4 Configuration Items

This section summarizes the symbols used in macro definitions for the static FlexCAN module configuration. These symbols can be put in the project-specific configuration file *appconfig.h* and are applied upon issuing the *FCAN_INIT* command.

Table 5-178. FlexCAN Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
FCAN_MCR_INIT	UWord16	Initial value for selected control bits of the FCMCR register. The bits are written to the FCMCR after the FlexCAN module is properly inintialized.	Reset value used, the FlexCAN module remains in HALT state,
FCAN_CTL0_INIT FCAN_CTL1_INIT\	UWord16	Initial value of the FlexCAN Control Registers (FCCTL0 and FCCTL1).	Register not initialized. Reset value used.
FCAN_RXGMASKL_INIT FCAN_RXGMASKH_INIT	UWord16	Initial value of the FlexCAN Receive Data Global Mask Registers (high and low part).	Register not initialized. Reset value used.
FCAN_RX14MASKL_INIT FCAN_RX14MASKH_INIT	UWord16	Initial value of the FlexCAN Receive Data Buffer 14 Mask Registers (high and low part).	Register not initialized. Reset value used.

Table 5-178. FlexCAN Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
FCAN_RX15MASKL_INIT FCAN_RX15MASKH_INIT	UWord16	Initial value of the FlexCAN Receive Data Buffer 15 Mask Registers (high and low part).	Register not initialized. Reset value used.
FCAN_IMASK1_INIT	UWord16	Initial value of the FCIMASK1 register.	Register not initialized. Reset value used.
FCAN_MAXMB_INIT	UWord16	Value to be written to the FCMAXMB register. This register contains several FlexCAN test bits so use it with care.	Register not initialized. Reset value used.

Note that for 56F8367 there are also valid symbols with FCAN2_ prefix corresponding to the FlexCAN2 module.

5.7.2.5 API Specification

This section briefly describes the API macros and functions.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call (macro) is generally represented in the following forms:

```
Word32 ioctl(const int *pModuleBase, void Cmd, void param);
Word16 ioctl(const int *pModuleBase, void Cmd, void param);
void ioctl(const int *pModuleBase, void Cmd, void param);
```

Description: The *ioctl* call "changes" FlexCAN device modes or accesses the FlexCAN register(s). Keep in mind that *ioctl* is treated as macro that is in result mostly compiled to an optimal inline code.

Arguments:

Table 5-179. FlexCAN Driver Arguments - ioctl

pModuleBase	in	FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is available only on 56F8367.
cmd	in	Command names found in *fcan.h.* See Table 5-180.

Table 5-179. FlexCAN Driver Arguments - ioctl

pParam	in, inout	Used to pass the relevant data to ioctl

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-180. FlexCAN Module ioctl Commands

Cmd	pParam	Return	Description
FCAN_INIT	NULL	None	Initializes the FlexCAN module using the *appconfig.h* values.
FCAN_STOP_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enter/leave low-power (STOP) mode.
FCAN_DEBUG_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enter/Leave HALT (Debug) mode.
FCAN_SOFT_RESET	NULL	None	Issue soft-reset of the FlexCAN module.
FCAN_SELF_WAKEUP_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enable/disable Self-Wakeup without CPU intervention.
FCAN_AUTO_PWRSAVE_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enable/disable Auto power save mode.
FCAN_TEST_READY	NULL	UWord16	Test whether the FlexCAN module is ready.
FCAN_TEST_DEBUG	NULL	UWord16	Test whether the FlexCAN module is in Freeze/HALT debug mode.
FCAN_TEST_STOP	NULL	UWord16	Test whether the FlexCAN module is in low-power STOP mode.
FCAN_INT_ENABLE	FCAN_BUSOFF_INT | FCAN_ERROR_INT | FCAN_WAKEUP_INT	None	Enable interrupts.
FCAN_INT_DISABLE	FCAN_BUSOFF_INT | FCAN_ERROR_INT | FCAN_WAKEUP_INT	None	Disable interrupts.
FCAN_LOOPBACK_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enable/disable test loopback mode.
FCAN_TIMER_SYNC_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enable/disable Timer Sync mode.

Table 5-180. FlexCAN Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
FCAN_LISTEN_ONLY_MODE	FCAN_ENABLE/ FCAN_DISABLE	None	Enable/disable Listen Only mode.
FCAN_SET_TX_FIRST_SCHEME	FCAN_LOWEST_ID/ FCAN_LOWEST_MB_NUMBER	None	Set transmit priority scheme.
FCAN_SET_SAMPLING	FCAN_1SAMP_PER_BIT / FCAN_3SAMPS_PER_BIT	None	Set number of hardware samples per bit.
FCAN_SET_PRESCALER	number 0 .. 255	None	Set PRES_DIV parameter.
FCAN_SET_RJW	FCAN_RJW_1 / FCAN_RJW_2 / FCAN_RJW_3 / FCAN_RJW_4 or UWord16 number 0..3	None	Set Re-Synchronization Jump Width (RJW) parameter.
FCAN_SET_PROP_SEG	FCAN_PROPSEG_1 / FCAN_PROPSEG_2 / FCAN_PROPSEG_3 / FCAN_PROPSEG_4 / FCAN_PROPSEG_5 / FCAN_PROPSEG_6 / FCAN_PROPSEG_7 / FCAN_PROPSEG_8 / or UWord16 number 0 .. 7	None	Set Propagation Segment (PROP_SEG) parameter.
FCAN_SET_PHASE_SEG1	FCAN_PSEG_1 / FCAN_PSEG_2 / FCAN_PSEG_3 / FCAN_PSEG_4 / or UWord16 number 0 .. 3	None	Set Phase Segment 1 (PHASE_SEG1) parameter.
FCAN_SET_PHASE_SEG2	FCAN_PSEG_1 / FCAN_PSEG_2 / FCAN_PSEG_3 / FCAN_PSEG_4 / or UWord16 number 0 .. 3	None	Set Phase Segment 2 (PHASE_SEG2) parameter.
FCAN_UNLOCK_ALL_MB	NULL	None	Unlock the one MB, which is currently locked by reading Free Running Timer (FRT).
FCAN_READ_ERR_AND_STATUS	NULL	UWord16	Read error and status register. Beware that the error bits are self-cleared after read.
FCAN_SET_MAXMB	UWord16 number 0 .. 15	None	Set maximum number of MB used (pParam+1).
FCAN_CLEAR_BOFF_INT	NULL	None	Clear BusOff interrupt status.
FCAN_CLEAR_ERR_INT	NULL	None	Clear Error interrupt status.

Table 5-180. FlexCAN Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
FCAN_CLEAR_WAKE_INT	NULL	None	Clear WakeUp interrupt status.
FCAN_CLEAR_INT	FCAN_STATUS_BOFF_INT | FCAN_STATUS_ERR_INT | FCAN_STATUS_WAKE_INT	None	Clear selected interrupt status bits.
FCAN_MBINT_ENABLE	FCAN_MBINT_0 | FCAN_MBINT_1 | FCAN_MBINT_2 | FCAN_MBINT_3 | FCAN_MBINT_4 | FCAN_MBINT_5 | FCAN_MBINT_6 | FCAN_MBINT_7 | FCAN_MBINT_8 | FCAN_MBINT_9 | FCAN_MBINT_10|FCAN_MBINT_11| FCAN_MBINT_12|FCAN_MBINT_13| FCAN_MBINT_14|FCAN_MBINT_15 or UWord16; one bit for each MB	None	Enable MB interrupts.
FCAN_MBINT_DISABLE	FCAN_MBINT_0 | FCAN_MBINT_1 | FCAN_MBINT_2 | FCAN_MBINT_3 | FCAN_MBINT_4 | FCAN_MBINT_5 | FCAN_MBINT_6 | FCAN_MBINT_7 | FCAN_MBINT_8 | FCAN_MBINT_9 | FCAN_MBINT_10|FCAN_MBINT_11| FCAN_MBINT_12|FCAN_MBINT_13| FCAN_MBINT_14|FCAN_MBINT_15 or UWord16; one bit for each MB	None	Disable MB interrupts.
FCAN_READ_MBINT_FLAGS	NULL	UWord16	Get MB interrupt source.
FCAN_CLEAR_MBINT_FLAGS	FCAN_MBINT_0 | FCAN_MBINT_1 | FCAN_MBINT_2 | FCAN_MBINT_3 | FCAN_MBINT_4 | FCAN_MBINT_5 | FCAN_MBINT_6 | FCAN_MBINT_7 | FCAN_MBINT_8 | FCAN_MBINT_9 | FCAN_MBINT_10|FCAN_MBINT_11| FCAN_MBINT_12|FCAN_MBINT_13| FCAN_MBINT_14|FCAN_MBINT_15 or UWord16; one bit for each MB	None	Clear MB interrupt flags.
FCAN_SET_RXGMASK FCAN_SET_RXGMASK_V	32bit mask value when MSB (bit31) is set, the mask is treated as for Extended ID OR the constant with FCAN_ID_EXT to set the MSB bit	None	Set Global RX mask, the passed mask is rebuilt to suit the register bit-scheme.
FCAN_SET_RX14MASK FCAN_SET_RX14MASK_V	32bit mask value when MSB (bit31) is set, the mask is treated as for Extended ID	None	Set MB14 RX mask, the passed mask is rebuilt to suit the register bit-scheme.
FCAN_SET_RX15MASK FCAN_SET_RX15MASK_V	32bit mask value when MSB (bit31) is set, the mask is treated as for Extended ID	None	Set MB15 RX mask, the passed mask is rebuilt to suit the register bit-scheme.

Table 5-180. FlexCAN Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
FCAN_SET_RXGMASK_RAW FCAN_SET_RX14MASK_RAW FCAN_SET_RX15MASK_RAW	32bit value written directly to the appropriate registers	None	Similar as above. The passed mask must already be in raw register format.
FCAN_GET_RX_ERR_COUNT	NULL	UWord16	Read RX error counter.
FCAN_GET_TX_ERR_COUNT	NULL	UWord16	Read TX error counter.
FCAN_GET_MB_MODULE	MB index 0..15	FCAN_MB*	Get pointer to MB module. The returned value can be used in subsequent ioctl calls using the FCANMB_xxx commands.

Table 5-181. FlexCAN Module ioctl Commands (Continued)

FCANMB_GET_ID	NULL	UWord32	Parse 29+1bit ID from the appropriate bits in given MB. Return as 32bit number where MSB signals Extended ID.
FCANMB_GET_ID_RAW	NULL	UWord32	Return raw 32bit ID part of the MB.
FCANMB_GET_LEN	NULL	UWord32	Return Data Frame Length of the specified MB.
FCANMB_GET_DATAPTR	NULL	UWord16 *	Return the pointer to Data part of the specified MB structure.
FCANMB_GET_CODE	NULL	UWord16	Return MB code/status field of the specified MB structure.
FCANMB_GET_TIMESTAMP	NULL	UWord16	Get MB TimeStamp, both bytes if available (Std. ID Frame received).
FCANMB_GET_TIMESTAMP8	NULL	UWord16	Get simplified MB TimeStamp (lower byte always zeroed).
FCANMB_SET_ID	32bit Id (constant) Use the bitwise OR of the number and FCAN_ID_EXT when setting extended ID. Use the FCAN_ID_RTR before sending the Remote Transmit Request.	None	Calculate the raw ID value and write it to the appropriate registers of given MB.
FCANMB_SET_ID_V	32bit Id (variable) Use the bitwise OR of the number and FCAN_ID_EXT when setting extended ID. Use the FCAN_ID_RTR before sending the Remote Transmit Request.	None	Calculate the raw ID value and write it to the appropriate registers of given MB.
FCANMB_SET_ID_RAW	32bit value	None	Write the raw ID value to the given MB.
FCANMB_SET_RTR	FCAN_ON/ FCAN_OFF	None	Set/Clear RTR bits in given MB. Can be used after the ID is written to MB.
FCANMB_SET_LEN	UWord16 number 0-8	None	Set length field of the MB.

FCANMB_SET_CODE	One of the code values. FCAN_MB_CODE_RXVOID / FCAN_MB_CODE_RXEMPTY / FCAN_MB_CODE_RXFULL / FCAN_MB_CODE_RXOVERRUN / FCAN_MB_CODE_RXBUSY / FCAN_MB_CODE_TXVOID / FCAN_MB_CODE_TXONCE / FCAN_MB_CODE_TXRALWAYS	None	Set code field of the MB.

The ioctl commands listed in Table 5-181 can be used to access the individual Message Buffer modules. The pModuleBase parameter of the ioctl call can be one of the predefined constants *FCAN_MB0...FCAN_MB15* (and *FCAN2_MB0...FCAN2_MB15* on 56F8367)or the pointer value returned by the *FCAN_GET_MB_MODULE* command.

5.7.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.7.3.1 FCAN_INIT - initialize FlexCAN module

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_INIT, NULL);
```

Arguments:

Table 5-182. FCAN_INIT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: This command initializes the FlexCAN module according to the values from *appconfig.h* configuration file. The module is put into the soft-reset state, then the control registers and control words of each MB are initialized. Also the global mask register and MB14-MB15 mask registers are initialized with the values from *appconfig.h* file.

FlexCAN module is then activated or deactivated by writing the Module Configuration Register (FCMCR). When the FCMCR initialization value is not given in *appconfig.h*, the FlexCAN module remains in post-reset "Debug" state. You have to set the FCMCR initialization value to 0x0000 for normal operation.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_INIT* ioctl command is implemented as a function call.

Example 5-161. FCAN_INIT

```
ioctl(FCAN, FCAN_INIT, NULL);
```

This code calls the driver initialization function.

5.7.3.2 FCAN_STOP_MODE - enter / leave low-power mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_STOP_MODE,
     UWord16 param);
```

Arguments:

Table 5-183. FCAN_STOP_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - enter low-power mode FCAN_DISABLE - leave low-power mode

Description: The *FCAN_STOP_MODE* ioctl command can be used to put the FlexCAN module into the low-power (STOP) mode. The low-power mode can be canceled by issuing the *FCAN_STOP_MODE* command again with parameter *FCAN_DISABLE* or by self wake-up feature of the FlexCAN. See *FCAN_SELF_WAKEUP_MODE* command for more information.

This command writes directly into the STOP bit of the FCMCR register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_STOP_MODE* ioctl command is implemented as a macro.

Example 5-162. FCAN_STOP_MODE

```
ioctl(FCAN, FCAN_STOP_MODE, FCAN_ENABLE);
```

This code enters the low-power sleep mode.

5.7.3.3 FCAN_DEBUG_MODE - enter / leave Freeze-Halt debug mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_DEBUG_MODE,
      UWord16 param);
```

Arguments:

Table 5-184. FCAN_DEBUG_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - enter the freeze-halt debug mode FCAN_DISABLE - leave debug mode

Description: After the *FCAN_DEBUG_MODE* ioctl command is issued, the FlexCAN module enters or leaves the Freeze/Halt debug mode.

This command writes directly into the FRZ1 and HALT bits of the FCMCR register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_DEBUG_MODE* ioctl command is implemented as a macro.

Example 5-163. FCAN_DEBUG_MODE

```
ioctl(FCAN, FCAN_DEBUG_MODE, FCAN_ENABLE);

while(! ioctl(FCAN, FCAN_TEST_DEBUG, NULL))
      ; /* wait for debug mode */
```

This code enters the debug mode and waits until it is acknowledged.

5.7.3.4 FCAN_SOFT_RESET - enter / leave the soft reset state

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SOFT_RESET, NULL);
```

Arguments:

Table 5-185. FCAN_SOFT_RESET ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_SOFT_RESET* ioctl command activates the internal soft-reset state of the FlexCAN module. Once soft-reset is entered, the module resets its internal state machines (sequencer, error counters, error flags, timer) and the host interface (FCMCR, ICR, TCR, FCIMASK and FCIFLAG). Other registers are no affected.

This command writes directly into the SOFT_RST bit of the FCMCR register. This bit is self-negated after the reset process finishes.

See FlexCAN module documentation for complete description of the soft-reset effects.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_SOFT_RESET* ioctl command is implemented as a macro.

Example 5-164. FCAN_SOFT_RESET

```
ioctl(FCAN, FCAN_SOFT_RESET, NULL);
```

This code activates the soft reset state of the FlexCAN module.

5.7.3.5 FCAN_SELF_WAKEUP_MODE - enable automatic recovery from STOP mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SELF_WAKEUP_MODE,
     UWord16 param);
```

Arguments:

Table 5-186. FCAN_SELF_WAKEUP_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - the module can be released from the low-power sleep mode without CPU intervention by detecting the recessive-to-dominant transition on the CAN bus. FCAN_DISABLE - only the CPU can release the module from the low-power sleep mode

Description: The *FCAN_SELF_WAKEUP_MODE* ioctl command enables FlexCAN to monitor the CAN bus while being in the low-power STOP mode. When self wake-up is enabled when entering the STOP mode by the FCAN_STOP_MODE command, the module will be looking for any recessive to dominant transition on the bus. Any such transition will then negate the STOP bit in the FCMCR and resume FlexCAN clocks.

This command writes directly into the SELF_WAKE bit of the FCMCR register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_SELF_WAKEUP_MODE* ioctl command is implemented as a macro.

Example 5-165. FCAN_SELF_WAKEUP_MODE

```
ioctl(FCAN, FCAN_SELF_WAKEUP_MODE, FCAN_ENABLE);
```

This code enables the self-wake up of the FlexCAN module.

5.7.3.6 FCAN_AUTO_PWRSAVE_MODE - enable / disable auto power save mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_AUTO_PWRSAVE_MODE,
     UWord16 param);
```

Arguments:

Table 5-187. FCAN_AUTO_PWRSAVE_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - the module automatically shut off its clocks to save power when it has no process to execute FCAN_DISABLE - module clocks are running normally

Description: The *FCAN_AUTO_PWRSAVE_MODE* ioctl command enables FlexCAN module to shut off its clocks to save power when it has no process to execute. Further, the clocks automatically resume when it has a task to execute, without any CPU intervention.

This command writes directly into the AUTO_POWER_SAVE bit of the FCMCR register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_AUTO_PWRSAVE_MODE* ioctl command is implemented as a macro.

Example 5-166. FCAN_AUTO_PWRSAVE_MODE

```
ioctl(FCAN, FCAN_AUTO_PWRSAVE_MODE, FCAN_ENABLE);
```

This code enables the auto power save feature of the FlexCAN module.

5.7.3.7 FCAN_TEST_READY - test module ready state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_TEST_READY, NULL);
```

Arguments:

Table 5-188. FCAN_TEST_READY ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_TEST_READY* ioctl command tests whether the FlexCAN module is either in low-power STOP mode or in Freeze/Halt debug mode. If neither of these modes are active, the ioctl returns non-zero value.

This command reads directly the NOT_RDY bit of the FCMCR register.

Returns: Non-zero if the module is ready and no sleep or debug mode are active. Zero when either sleep or debug more are currently active.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_TEST_READY* ioctl command is implemented as a macro.

Example 5-167. FCAN_TEST_READY

```
if ( ioctl(FCAN, FCAN_TEST_READY, NULL) )
{
      ...
}
```

This code test whether the FlexCAN module is ready.

5.7.3.8 FCAN_TEST_DEBUG - test module debug state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_TEST_DEBUG, NULL);
```

Arguments:

Table 5-189. FCAN_TEST_DEBUG ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_TEST_DEBUG* ioctl command tests whether the FlexCAN module is in debug mode. The ioctl returns non-zero when the debug mode is active.

This command reads directly the FREEZ_ACK bit of the FCMCR register.

Returns: Non-zero if FlexCAN is currently in debug mode. Otherwise the return value is zero.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_TEST_DEBUG* ioctl command is implemented as a macro.

Example 5-168. FCAN_TEST_DEBUG

```
if( ioctl(FCAN, FCAN_TEST_DEBUG, NULL) )
{
      ...
}
```

This command tests whether the module is in debug mode.

5.7.3.9 FCAN_TEST_STOP - test module low-power state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_TEST_STOP, NULL);
```

Arguments:

Table 5-190. FCAN_TEST_STOP ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_TEST_STOP* ioctl command tests whether the FlexCAN module is in low-power STOP mode. The ioctl returns non-zero when the low-power mode is active.

This command reads directly the STOP_ACK bit of the FCMCR register.

Returns: Non-zero if FlexCAN is currently in low-power stop mode. Otherwise the return value is zero.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_TEST_STOP* ioctl command is implemented as a macro.

Example 5-169. FCAN_TEST_STOP

```
if( ioctl(FCAN, FCAN_TEST_STOP, NULL) )
{
      ...
}
```

This command tests whether the module is in stop mode.

5.7.3.10 FCAN_INT_ENABLE - enable FlexCAN interrupts

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_INT_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-191. FCAN_INT_ENABLE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The combination of bits for each interrupt to be enabled. Use the OR-combination of these predefined constants: FCAN_BUSOFF_INT - bus off interrupt FCAN_ERROR_INT - error interrupt FCAN_WAKEUP_INT - wake up interrupt

Description: The *FCAN_INT_ENABLE* ioctl command enables the selected interrupts to be signalled to the core. See also *FCAN_MBINT_ENABLE* command.

This command directly writes the WAKE_MASK bit of the FCMCR register and the BUSOFF_MASK and ERROR_MASK bits in the Control Register 0 (FCCTL0).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-170. FCAN_INT_ENABLE

```
ioctl(FCAN, FCAN_INT_ENABLE, FCAN_ERROR_INT);
```

This code enables the error interrupt to be signalled to the MCU core.

5.7.3.11 FCAN_INT_DISABLE - disable FlexCAN interrupts

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_INT_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-192. FCAN_INT_DISABLE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The combination of bits for each interrupt to be disabled. Use the OR-combination of these predefined constants: FCAN_BUSOFF_INT - bus off interrupt FCAN_ERROR_INT - error interrupt FCAN_WAKEUP_INT - wake up interrupt

Description: The *FCAN_INT_DISABLE* ioctl command disables (masks) the selected interrupts.

This command directly writes the WAKE_MASK bit of the FCMCR register and the BUSOFF_MASK and ERROR_MASK bits in the Control Register 0 (FCCTL0).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-171. FCAN_INT_DISABLE

```
ioctl(FCAN, FCAN_INT_DISABLE, FCAN_ERROR_INT | FCAN_WAKEUP_INT);
```

This code disables the wake-up and error interrupts to be signalled to the MCU core.

5.7.3.12 FCAN_LOOPBACK_MODE - enable / disable test loopback mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_LOOPBACK_MODE,
     UWord16 param);
```

Arguments:

Table 5-193. FCAN_LOOPBACK_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - loopback mode enabled FCAN_DISABLE - loopback mode disabled, normal operation

Description: The *FCAN_LOOPBACK_MODE* ioctl command sets or resets the internal loopback mode of the FlexCAN module. When this mode is set, the internal loopback is enabled and the module is prevented to see bus activity.

This command writes directly the TEST_EN bit in the FCMAXMB register and LOOPB bit int he FCCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_LOOPBACK_MODE* ioctl command is implemented as a macro.

Example 5-172. FCAN_LOOPBACK_MODE

```
ioctl(FCAN, FCAN_LOOPBACK_MODE, FCAN_DISABLE);
```

This code disables the internal loopback mode of the FlexCAN module.

5.7.3.13 FCAN_TIMER_SYNC_MODE - enable / disable timer synchronization mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_TIMER_SYNC_MODE,
     UWord16 param);
```

Arguments:

Table 5-194. FCAN_TIMER_SYNC_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - timer synchronization mode enabled FCAN_DISABLE -timer synchronization mode disabled

Description: The *FCAN_TIMER_SYNC_MODE* ioctl command enables or disables the timer synchronization mode of the FlexCAN module. Once enabled, the FlexCAN Free Running Timer is reset (cleared) each time the message is received in the Message Buffer 0. This feature enables the time synchronization between multiple FlexCAN stations with a special message.

This command directly accesses the TSYNC bit of the FCCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_TIMER_SYNC_MODE* ioctl command is implemented as a macro.

Example 5-173. FCAN_TIMER_SYNC_MODE

```
ioctl(FCAN, FCAN_TIMER_SYNC_MODE, FCAN_DISABLE);
```

This code disables the timer synchronization feature of the FlexCAN module.

5.7.3.14 FCAN_LISTEN_ONLY_MODE - enable / disable timer synchronization mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_LISTEN_ONLY_MODE,
     UWord16 param);
```

Arguments:

Table 5-195. FCAN_LISTEN_ONLY_MODE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ENABLE - the module is in Listen-Only mode FCAN_DISABLE -the module is in normal mode

Description: The *FCAN_LISTEN_ONLY_MODE* ioctl command enables or disables the listen-only mode of the FlexCAN module. Once this mode is enabled, the FlexCAN is able to receive messages without acknowledging, what might be useful for diagnostic purposes.

This command directly accesses the LOM bit of the FCCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_LISTEN_ONLY_MODE* ioctl command is implemented as a macro.

Example 5-174. FCAN_LISTEN_ONLY_MODE

```
ioctl(FCAN, FCAN_LISTEN_ONLY_MODE, FCAN_DISABLE);
```

This code disables the listen-only mode of the FlexCAN module.

5.7.3.15 FCAN_SET_TX_FIRST_SCHEME - set transmit priority scheme

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_TX_FIRST_SCHEME,
     UWord16 param);
```

Arguments:

Table 5-196. FCAN_SET_TX_FIRST_SCHEME ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_LOWEST_ID - the Message Buffer with lowest CAN frame ID is transmitted first FCAN_LOWEST_MB_NUMBER - the lowest number MB is transmitted first

Description: The *FCAN_SET_TX_FIRST_SCHEME* ioctl command configures the transmission priority scheme. There are two priority schemes to choose from:

- The Message Buffer which is ready to transmit the CAN frame with the lowest ID is transmitted first
- The lowest number (index) Message Buffer is transmitted first

This command directly accesses the LBUF bit of the FCCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_SET_TX_FIRST_SCHEME* ioctl command is implemented as a macro.

Example 5-175. FCAN_SET_TX_FIRST_SCHEME

```
ioctl(FCAN, FCAN_SET_TX_FIRST_SCHEME, FCAN_LOWEST_ID);
```

This code sets enables the lowest ID frames to be transmitted first.

5.7.3.16 FCAN_SET_SAMPLING - set bit-sampling mode

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_SAMPLING,
     UWord16 param);
```

Arguments:

Table 5-197. FCAN_SET_SAMPLING ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_1SAMP_PER_BIT - one sample per bit FCAN_3SAMPS_PER_BIT - three samples per bit

Description: The *FCAN_SET_SAMPLING* ioctl command configures the receiver machine in terms how many times each bit is sampled. The possible values are “once per bit” (reset default) or “three times per bit”. Three times sampling includes the regular sample point and two preceding samples and uses majority rule to decide the resulting bit value.

This command directly accesses the SAMP bit of the FCCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_SET_SAMPLING* ioctl command is implemented as a macro.

Example 5-176. FCAN_SET_SAMPLING

```
ioctl(FCAN, FCAN_SET_SAMPLING, FCAN_3SAMPS_PER_BIT);
```

This code selects the three-samples-per-bit sampling mode.

5.7.3.17 FCAN_SET_PRESCALER - set prescaler divide factor

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_PRESCALER,
     UWord16 param);
```

Arguments:

Table 5-198. FCAN_SET_PRESCALER ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The value to set the prescaler to (0...255)

Description: The *FCAN_SET_PRESCALER* ioctl command sets the ratio between the system clock frequency and the serial clock of the FlexCAN module. The serial clock (sclock) then defines the time quantum for the other timing parameters of the FlexCAN module.

1 sclock = 1 time quantum

This command directly accesses the PRES_DIV bit-field of the FCCTL1 register. The value passed to the ioctl command is directly written to the bit-field so it accepts the values between 0 and 255. The system clock and serial clock ratio is then PRES_DIV+1.

Returns: None.

Range Issues: Only values 0 to 255 are allowed.

Special Issues: The resulting baud rate of the FlexCAN module is dependent on several other parameters in FCCTL0 and FCCTL1 registers. Section 5.7.2.1 contains more information about bit-timing topic.

Design/Implementation: The *FCAN_SET_PRESCALER* ioctl command is implemented as a macro.

Example 5-177. FCAN_SET_PRESCALER

```
ioctl(FCAN, FCAN_SET_PRESCALER, 7);
```

This code selects the clock divisor to 8. In the case the 8 MHz external oscillator is attached to the MCU, the serial clock will run at 1 MHz and one time quantum will be equal to 1 us.

5.7.3.18 FCAN_SET_RJW - set Re-synchronization Jump Width

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_RJW,
     UWord16 param);
```

Arguments:

Table 5-199. FCAN_SET_RJW ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The value to set the Re-Synchronization Jump Width parameter to. Use the following predefined constants or their UWord16 numeric representation given in parenthesis. FCAN_RJW_1 (0) FCAN_RJW_2 (1) FCAN_RJW_3 (2) FCAN_RJW_4 (3)

Description: The *FCAN_SET_RJW* ioctl command sets the Re-Synchronization Jump Width parameter of the CAN bit timing machine. The RJW is specified in time-quanta units and is always at least 1 tq. See the CAN standard document for more information.

This command writes the given parameter directly to the RJW bit-field of the FCCTL1 register. The value of Re-Synchronization Jump Width parameter is equal to FCCTL1.RJW+1.

Returns: None.

Range Issues: Numerical values 0 to 3 are allowed only, use the *FCAN_RJW_n* constants defined in the *fcan.h* file.

Special Issues: The resulting baud rate of the FlexCAN module is dependent on several other parameters in FCCTL0 and FCCTL1 registers. Section 5.7.2.1 contains more information about this topic.

Design/Implementation: The *FCAN_SET_RJW* ioctl command is implemented as a macro.

Example 5-178. FCAN_SET_RJW

```
ioctl(FCAN, FCAN_SET_RJW, FCAN_RJW_1);
```

This code sets the Re-synchronization Jump Width parameter to 1.

5.7.3.19 FCAN_SET_PROP_SEG - set Propagation Segment

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_PROP_SEG,
     UWord16 param);
```

Arguments:

Table 5-200. FCAN_SET_PROP_SEG ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The value to set the Propagation Segment to. Use the following constants or their UWord16 numeric representation given in parenthesis. FCAN_PROPSEG_1 (0) FCAN_PROPSEG_2 (1) FCAN_PROPSEG_3 (2) FCAN_PROPSEG_4 (3) FCAN_PROPSEG_5 (4) FCAN_PROPSEG_6 (5) FCAN_PROPSEG_7 (6) FCAN_PROPSEG_8 (7)

Description: The *FCAN_SET_PROP_SEG* ioctl command sets the Propagation Segment parameter of the CAN bit timing machine. The value is specified in time-quanta units and must be always at least 1 tq. See CAN standard for more information.

This command writes the given parameter directly to the PROPSEG bit-field of the FCCTL0 register. The value of Propagation Segment parameter is then equal to FCCTL0.PROPSEG+1.

Returns: None.

Range Issues: Numerical values 0 to 7 are allowed only, use the *FCAN_PROPSEG_n* constants defined in the *fcan.h* file.

Special Issues: The resulting baud rate of the FlexCAN module is dependent on several other parameters in FCCTL0 and FCCTL1 registers. Section 5.7.2.1 contains more information about this topic.

Design/Implementation: The *FCAN_SET_PROP_SEG* ioctl command is implemented as a macro.

Example 5-179. FCAN_SET_PROP_SEG

```
ioctl(FCAN, FCAN_SET_PROP_SEG, FCAN_PROPSEG_4);
```

This code sets the propagation segment to 4.

5.7.3.20 FCAN_SET_PHASE_SEG1, FCAN_SET_PHASE_SEG2 - set Phase Buffer Segments

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_PHASE_SEG1,
     UWord16 param);
void ioctl(const int *pModuleBase, FCAN_SET_PHASE_SEG2,
     UWord16 param);
```

Arguments:

Table 5-201. FCAN_SET_PHASE_SEG1, FCAN_SET_PHASE_SEG2 ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The value to set the Phase Buffer Segment to. Use the following constants or their UWord16 numeric representation given in parenthesis. FCAN_PSEG_1 (0) FCAN_PSEG_2 (1) FCAN_PSEG_3 (2) FCAN_PSEG_4 (3) FCAN_PSEG_5 (4) FCAN_PSEG_6 (5) FCAN_PSEG_7 (6) FCAN_PSEG_8 (7)

Description: The *FCAN_SET_PHASE_SEG1* and *FCAN_SET_PHASE_SEG2* ioctl commands set the Phase Buffer Segment parameters of the CAN bit timing machine. The parameter value is specified in time-quanta units and must be always at least 1 tq. See CAN standard for more information.

This command writes the given parameter directly to the PSEG1 or PSEG2 bit-fields of the FCCTL1 register. The value of Phase Buffer Segment parameter is then equal to FCCTL1.PSEG1+1 or FCCTL1.PSEG2+1 respectively.

Returns: None.

Range Issues: Numerical values 0 to 7 are allowed only, use the *FCAN_PSEG_n* constants defined in the *fcan.h* file.

Special Issues: The resulting baud rate of the FlexCAN module is dependent on several other parameters in FCCTL0 and FCCTL1 registers. Section 5.7.2.1 contains more information about this topic.

Design/Implementation: The *FCAN_SET_PHASE_SEG1* ioctl command is implemented as a macro.

Example 5-180. FCAN_SET_PHASE_SEG1, FCAN_SET_PHASE_SEG2

```
ioctl(FCAN, FCAN_SET_PHASE_SEG1, FCAN_PSEG_4);
```

This code sets the Phase Buffer Segment 1 to 4.

5.7.3.21 FCAN_UNLOCK_ALL_MB - unlock the MB currently locked

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_UNLOCK_ALL_MB, NULL);
```

Arguments:

Table 5-202. FCAN_UNLOCK_ALL_MB ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_UNLOCK_ALL_MB* ioctl command unlocks the internal MB-locking mechanism of the FlexCAN module. Normally, the MB is locked when control-status word of the received MB is read. Locked MB then can not be overwritten by the FlexCAN module so the data coherency is assured when CPU reads the MB data part. To unlock the locked MB, either any other MB should be locked or the CPU should issue the read instruction of the FCTMR register.

This command compiles to the assembly instruction which reads the FCTMR register, so any locked MB gets unlocked. It is assured that the C-compiler optimizer does not cut this instruction out of the application even if the value read from the timer is not used.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_UNLOCK_ALL_MB* ioctl command is implemented as a macro.

Example 5-181. FCAN_UNLOCK_ALL_MB

```
ioctl (FCAN, FCAN_UNLOCK_ALL_MB, NULL);
```

This code unlocks any MB currently locked.

5.7.3.22 FCAN_SET_MAXMB - set maximum number of MB used

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_MAXMB, UWord16 param);
```

Arguments:

Table 5-203. FCAN_SET_MAXMB ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The value to set the maximum number of MB used.

Description: The *FCAN_SET_MAXMB* ioctl command sets the maximum number of MB used.

This command writes the given parameter directly to the MAXMB bit-field of the FCMAXMB register. The number of MB in use is then equal to FCMAXMB.MAXMB+1.

Returns: None.

Range Issues: Numerical values 0 to 15 are allowed only.

Special Issues: None.

Design/Implementation: The *FCAN_SET_MAXMB* ioctl command is implemented as a macro.

Example 5-182. FCAN_SET_MAXMB

```
ioctl(FCAN, FCAN_SET_MAXMB, 5);
```

This code sets the number of MB in use to 6.

5.7.3.23 FCAN_READ_ERR_AND_STATUS - read error and status bits

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_READ_ERR_AND_STATUS,
      NULL);
```

Arguments:

Table 5-204. FCAN_READ_ERR_AND_STATUS ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_READ_ERR_AND_STATUS* ioctl command reads the FCSTATUS register of the FlexCAN module. The returned value can be tested using the following bit name macros:

```
FCAN_STATUS_BIT1_ERR
FCAN_STATUS_BIT0_ERR
FCAN_STATUS_ACK_ERR
FCAN_STATUS_CRC_ERR
FCAN_STATUS_FORM_ERR
FCAN_STATUS_STUFF_ERR
FCAN_STATUS_TX_WARN
FCAN_STATUS_RX_WARN
FCAN_STATUS_IDLE
FCAN_STATUS_TXRX
FCAN_STATUS_FCS_MASK
FCAN_STATUS_FCS_BUSOFF
FCAN_STATUS_FCS_EACTIVE
FCAN_STATUS_FCS_EPASSIVE
FCAN_STATUS_BOFF_INT
FCAN_STATUS_ERR_INT
FCAN_STATUS_WAKE_INT
```

Returns: FCSTATUS register value.

Range Issues: None.

Special Issues: Some of the bits in the FCSTATUS register are self-cleared after read.

Design/Implementation: The *FCAN_READ_ERR_AND_STATUS* ioctl command is implemented as a macro.

Example 5-183. FCAN_READ_ERR_AND_STATUS

```
UWord16 status = ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);

if ( status & FCAN_STATUS_FCS_BUSOFF )
{
      ...
}
```

This code tests the bus-off condition.

5.7.3.24 FCAN_CLEAR_BOFF_INT - clear “Bus Off” interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_CLEAR_BOFF_INT, NULL);
```

Arguments:

Table 5-205. FCAN_CLEAR_BOFF_INT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_CLEAR_BOFF_INT* ioctl command clears the “Bus Off” interrupt flag in the FCSTATUS register. This bit is set any time the FlexCAN state changes to “bus off”. When an appropriate interrupt is not masked, the “Bus-Off” interrupt is generated.

The *FCAN_CLEAR_BOFF_INT* command can be used in the interrupt service routine to acknowledge the interrupt.

Returns: None.

Range Issues: None.

Special Issues: The Error and Status Register must be read before clearing the flag.

Design/Implementation: The *FCAN_CLEAR_BOFF_INT* ioctl command is implemented as a macro.

Example 5-184. FCAN_CLEAR_BOFF_INT

```
UWord16 status = ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
...
ioctl(FCAN, FCAN_CLEAR_BOFF_INT, NULL);
```

This code clears the “Bus Off” interrupt flag.

5.7.3.25 FCAN_CLEAR_ERR_INT - clear “Error” interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_CLEAR_ERR_INT, NULL);
```

Arguments:

Table 5-206. FCAN_CLEAR_ERR_INT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_CLEAR_ERR_INT* ioctl command clears the Error interrupt flag in the FCSTATUS register. This bit is set any time one of the FlexCAN error STATUS bits is set. When appropriate (error) interrupt is not masked, the interrupt is generated.

The *FCAN_CLEAR_ERR_INT* command can be used in the interrupt service routine to acknowledge the interrupt.

Returns: None.

Range Issues: None.

Special Issues: The Error and Status Register must be read before clearing the flag.

Design/Implementation: The *FCAN_CLEAR_ERR_INT* ioctl command is implemented as a macro.

Example 5-185. FCAN_CLEAR_ERR_INT

```
UWord16 status = ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
...
ioctl(FCAN, FCAN_CLEAR_ERR_INT, NULL);
```

This code clears the Error interrupt flag.

5.7.3.26 FCAN_CLEAR_WAKE_INT - clear “Wake Up” interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_CLEAR_WAKE_INT, NULL);
```

Arguments:

Table 5-207. FCAN_CLEAR_WAKE_INT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_CLEAR_WAKE_INT* ioctl command clears the “Wake Up” interrupt flag in the FCSTATUS register. This bit is set when FlexCAN is in low-power STOP mode and any recessive to dominant transition is detected on the CAN bus. When appropriate (wake up) interrupt is not masked, the interrupt is generated.

The *FCAN_CLEAR_WAKE_INT* command can be used in the interrupt service routine to acknowledge the interrupt.

Returns: None.

Range Issues: None.

Special Issues: The Error and Status Register must be read before clearing the flag.

Design/Implementation: The *FCAN_CLEAR_WAKE_INT* ioctl command is implemented as a macro.

Example 5-186. FCAN_CLEAR_WAKE_INT

```
UWord16 status = ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
...
ioctl(FCAN, FCAN_CLEAR_WAKE_INT, NULL);
```

This code clears the Wake Up interrupt flag.

5.7.3.27 FCAN_CLEAR_INT - clear interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_CLEAR_INT,
      UWord16 param);
```

Arguments:

Table 5-208. FCAN_CLEAR_INT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Interrupt flags to be cleared. Use the OR-combination of the following constants: FCAN_STATUS_BOFF_INT | FCAN_STATUS_ERR_INT | FCAN_STATUS_INT |

Description: The *FCAN_CLEAR_INT* ioctl command clears the selected interrupt flags in the FCSTATUS register. Unlike with the *FCAN_CLEAR_BOFF_INT*, *FCAN_CLEAR_ERR_INT* and *FCAN_CLEAR_WAKE_INT* commands, any combination of interrupt flags can be cleared.

Returns: None.

Range Issues: None.

Special Issues: The Error and Status Register must be read before clearing the flag(s).

Design/Implementation: The *FCAN_CLEAR_INT* ioctl command is implemented as a macro.

Example 5-187. FCAN_CLEAR_INT

```
UWord16 status = ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
...
ioctl(FCAN, FCAN_CLEAR_INT,
      FCAN_STATUS_BOFF_INT | FCAN_STATUS_ERR_INT);
```

This code clears the “Bus Off” and Error interrupt flags.

5.7.3.28 FCAN_MBINT_ENABLE - enable Message Buffer interrupts

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_MBINT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-209. FCAN_MBINT_ENABLE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Combination of bits. Each bit represents one of the 16 Message Buffers of the FlexCAN module. You can also use the following predefined constants: FCAN_MBINT_0 FCAN_MBINT_1 ... FCAN_MBINT_15

Description: The *FCAN_MBINT_ENABLE* ioctl command enables the interrupt generation for the selected Message Buffers. The MB interrupts are generated after each successful transmission of reception of the buffer. All the Message Buffers share a single interrupt vector. The command modifies content of the FCIMASK1 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_MBINT_ENABLE* ioctl command is implemented as a macro.

Example 5-188. FCAN_MBINT_ENABLE

```
ioctl(FCAN, FCAN_MBINT_ENABLE, FCAN_MBINT_0 | FCAN_MBINT_1);
```

This code enables interrupts for Message Buffers 0 and 1.

5.7.3.29 FCAN_MBINT_DISABLE - disable Message Buffer interrupts

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_MBINT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-210. FCAN_MBINT_DISABLE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Combination of bits. Each bit represents one of the 16 Message Buffers of the FlexCAN module. You can also use the following predefined constants: FCAN_MBINT_0 FCAN_MBINT_1 ... FCAN_MBINT_15

Description: The *FCAN_MBINT_DISABLE* ioctl command disables the interrupts for the selected Message Buffers. The command modifies content of the FCIMASK1 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_MBINT_DISABLE* ioctl command is implemented as a macro.

Example 5-189. FCAN_MBINT_DISABLE

```
ioctl(FCAN, FCAN_MBINT_DISABLE, FCAN_MBINT_15 | FCAN_MBINT_0);
```

This code disables interrupts for Message Buffers 0 and 15.

5.7.3.30 FCAN_READ_MBINT_FLAGS - get the MB interrupt sources

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_READ_MBINT_FLAGS,
      NULL);
```

Arguments:

Table 5-211. FCAN_READ_MBINT_FLAGS ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_READ_MBINT_FLAGS* ioctl command reads the FCIFLAG1 register to get the list of Message Buffers (each bit for one MB) which generated the interrupt - either after successful transmission or reception.

Returns: 16 bit value, each bit set represents one MB which requires attention in the interrupt service routine.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_READ_MBINT_FLAGS* ioctl command is implemented as a macro.

Example 5-190. FCAN_READ_MBINT_FLAGS

```
UWord16 mbInt = ioctl(FCAN, FCAN_READ_MBINT_FLAGS, NULL);

if(mbInt & FCAN_MBINT_3)
{
      ...
}
```

This code reads and stores the list of Message Buffers for which the reception or transmission finished. The MB3 bit is then tested in the returned value.

5.7.3.31 FCAN_CLEAR_MBINT_FLAGS - clear MB interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_CLEAR_MBINT_FLAGS,
           UWord16 param);
```

Arguments:

Table 5-212. FCAN_CLEAR_MBINT_FLAGS ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Any combination of bits representing Message Buffers for which the interrupts are to be acknowledged (flags cleared). You can also use the following predefined constants: FCAN_MBINT_0 FCAN_MBINT_1 ... FCAN_MBINT_15

Description: The *FCAN_CLEAR_MBINT_FLAGS* ioctl command clears the selected interrupt flags and acknowledges the MB interrupt. In a standard interrupt service routine, the value read by the FCAN_READ_MBINT_FLAGS should be passed to this command as the parameter to assure that only previously detected interrupts are acknowledged. However, the FlexCAN module follows the SRSv2.0 when clearing the interrupt bits. The standard requires any flag must be first first read as one before it can be cleared (by writing one). This is why one can pass the 0xffff constant to the FCAN_CLEAR_MBINT_FLAGS while still being sure that only interrupts read from the FCIFLAG1 register as one get acknowledged (cleared).

Returns: None.

Range Issues: None.

Special Issues: The Interrupt Flag Register 1 must be read before clearing the flag.

Design/Implementation: The *FCAN_CLEAR_MBINT_FLAGS* ioctl command is implemented as a macro.

Example 5-191. FCAN_CLEAR_MBINT_FLAGS

```
UWord16 mbInt = ioctl(FCAN, FCAN_READ_MBINT_FLAGS, NULL);

... // process Message Buffers identified in mbInt

ioctl(FCAN, FCAN_CLEAR_MBINT_FLAGS, 0xffff);
```

This code first reads the interrupt sources, and then clears all the flags previously read as one.

5.7.3.32 FCAN_SET_RXGMASK, FCAN_SET_RXGMASK_V - set global receiver mask

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_RXGMASK,
           UWord32 param);
void ioctl(const int *pModuleBase, FCAN_SET_RXGMASK_V,
           UWord32 param);
```

Arguments:

Table 5-213. FCAN_SET_RXGMASK ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Mask value. The most significant bit (bit31), when set, identifies the extended (29bit) mask identifier as defined by CAN 2.0B standard. When MSB is not set, the standard 11bit mask identifier is used.

Description: The *FCAN_SET_RXGMASK* ioctl command sets the Receive Data Global Mask value. The mask bits are applied to all receiving Message Buffers excluding MB 14 and 15. The value passed to this ioctl command should contain information about whether it is specified in standard 11bit format or extended 29bit format. The most significant bit (MSB) is reserved for this purpose. When MSB is set, the value is taken as extended identifier mask. You can use the FCAN_ID_EXT constant to set the MSB bit of the value. Several examples of composing the identifier and mask values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567 (= 0x1234567 | FCAN_ID_EXT)
0x80000033   ... Extended identifier 0x33 (= 0x33 | FCAN_ID_EXT)
```

The *FCAN_SET_RXGMASK* ioctl command writes directly into the FCRxGMASK_H and FCRxGMASK_L FlexCAN registers. The mask value passed as the command parameter is first converted to a raw format suitable for writing to the registers. The inline function *FCAN_Id2Idr()* is used for this conversion.

Returns: None.

Range Issues: Standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. Extended identifiers (MSB=1) are limited to 29 bits so the valid range is 0x8000000...0x9fffffff.

Special Issues: None.

Design/Implementation: The *FCAN_SET_RXGMASK* ioctl command is implemented as a macro. The *FCAN_SET_RXGMASK_V* ioctl command is implemented as a function call.

Example 5-192. FCAN_SET_RXGMASK

```
ioctl(FCAN, FCAN_SET_RXGMASK, 0xff);
```

This code sets the global receiver mask to 0xff.

5.7.3.33 FCAN_SET_RXGMASK_RAW - set global receiver mask

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_RXGMASK_RAW,
           UWord32 param);
```

Arguments:

Table 5-214. FCAN_SET_RXGMASK_RAW ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Raw mask value. This value must be in the format suitable for Flex-CAN mask registers (high and low).

Description: The *FCAN_SET_RXGMASK_RAW* ioctl command sets the Receive Data Global Mask value. The mask bits are applied to all receiving Message Buffers excluding MB 14 and 15. Unlike with the FCAN_SET_RXGMASK ioctl command, the value passed to FCAN_SET_RXGMASK_RAW command should already be converted to a format suitable for writing into FCRxGMASK_H and FCRxGMASK_L FlexCAN registers. The inline function *FCAN_Id2Idr()* can be used for this conversion.

Returns: None.

Range Issues: None.

Special Issues: See FlexCAN documentation for the details about mask registers format.

Design/Implementation: The *FCAN_SET_RXGMASK_RAW* ioctl command is implemented as a macro.

Example 5-193. FCAN_SET_RXGMASK_RAW

```
ioctl(FCAN, FCAN_SET_RXGMASK_RAW, 0xff << 21);
```

This code sets the global receiver mask to 0xff.

5.7.3.34 FCAN_SET_RX14MASK, FCAN_SET_RX15MASK, FCAN_SET_RX14MASK_V, FCAN_SET_RX15MASK_V - set receiver mask for MB14 and MB15

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_RX14MASK,
           UWord32 param);
void ioctl(const int *pModuleBase, FCAN_SET_RX15MASK,
           UWord32 param);
void ioctl(const int *pModuleBase, FCAN_SET_RX14MASK_V,
           UWord32 param);
void ioctl(const int *pModuleBase, FCAN_SET_RX15MASK_V,
           UWord32 param);
```

Arguments:

Table 5-215. FCAN_SET_RX14MASK, FCAN_SET_RX15MASK ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Mask value. The most significant bit (bit31), when set, identifies the extended (29bit) mask identifier as defined by CAN 2.0B standard. When MSB is not set, the standard 11bit mask identifier is used.

Description: The *FCAN_SET_RX14MASK* and *FCAN_SET_RX15MASK* ioctl commands set the Receive Data Masks for Message Buffer 14 or Message Buffer 15 respectively. The value passed to these ioctl commands should contain information about whether it is specified in standard 11bit format or extended 29bit format. The most significant bit (MSB) is reserved for this purpose. When MSB is set, the value is taken as extended identifier mask. Several examples of composing the identifier and mask values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567 (= 0x1234567 | FCAN_ID_EXT)
0x80000033   ... Extended identifier 0x33 (= 0x33 | FCAN_ID_EXT)
```

The *FCAN_SET_RX14MASK* ioctl command writes directly into the FCRx14MASK_H and FCRx14MASK_L FlexCAN registers. The *FCAN_SET_RX15MASK* ioctl command writes directly into the FCRx15MASK_H and FCRx15MASK_L FlexCAN registers. The mask value passed as the command parameter is first converted to a raw format suitable for writing to the registers. The inline function *FCAN_Id2Idr()* is used for this conversion.

Returns: None.

Range Issues: Standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. Extended identifiers (MSB=1) are limited to 29 bits so the valid range is 0x8000000...0x9fffffff.

Special Issues: None.

Design/Implementation: Both the *FCAN_SET_RX14MASK* and *FCAN_SET_RX15MASK* ioctl commands are implemented as a macro. Both the *FCAN_SET_RX14MASK_V* and *FCAN_SET_RX15MASK_V* ioctl commands are implemented as a function call.

Example 5-194. FCAN_SET_RX14MASK, FCAN_SET_RX15MASK

```
ioctl(FCAN, FCAN_SET_RX14MASK, 0x0f);
```

This code sets the receiver mask of Message Buffer 14 to 0x0f.

5.7.3.35 FCAN_SET_RX14MASK_RAW, FCAN_SET_RX15MASK_RAW - set receiver mask for MB15 and MB15

Call(s):

```
void ioctl(const int *pModuleBase, FCAN_SET_RX14MASK_RAW,
           UWord32 param);
void ioctl(const int *pModuleBase, FCAN_SET_RX15MASK_RAW,
           UWord32 param);
```

Arguments:

Table 5-216. FCAN_SET_RX14MASK_RAW, FCAN_SET_RX15MASK_RAW ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	Raw mask value. This value must be in the format suitable for Flex-CAN mask registers (high and low).

Description: The *FCAN_SET_RX14MASK_RAW* and *FCAN_SET_RX15MASK_RAW* ioctl commands set the Receive Data Mask values for Message Buffer 14 or Message Buffer 15 respectively. Unlike with the FCAN_SET_RX14MASK or FCAN_SET_RX15MASK ioctl commands, the value passed to these commands should already be converted to a format suitable for writing into FCRx14MASK_H and FCRx14MASK_L (or FCRx15MASK_H and FCRx15MASK_L) FlexCAN registers. The inline function *FCAN_Id2Idr()* can be used for this conversion.

Returns: None.

Range Issues: None.

Special Issues: See FlexCAN documentation for the details about mask registers format.

Design/Implementation: The *FCAN_SET_RX14MASK_RAW* and *FCAN_SET_RX15MASK_RAW* ioctl commands are implemented as a macro.

Example 5-195. FCAN_SET_RX14MASK_RAW, FCAN_SET_RX15MASK_RAW

```
ioctl(FCAN, FCAN_SET_RX14MASK_RAW, 0xff << 21);
```

This code sets the receiver mask of Message Buffer 14 to 0xff.

5.7.3.36 FCAN_GET_RX_ERR_COUNT - read RX error counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_GET_RX_ERR_COUNT,
          NULL);
```

Arguments:

Table 5-217. FCAN_GET_RX_ERR_COUNT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_GET_RX_ERR_COUNT* ioctl command returns the 8bit value of the FlexCAN RX error counter. See the FlexCAN documentation for detailed description of error counter values.

Returns: Content of the RX error counter field of the error counter register (FC_ERR_CNTRS).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_GET_RX_ERR_COUNT* ioctl command is implemented as a macro.

Example 5-196. FCAN_GET_RX_ERR_COUNT

```
UWord16 rxErr;

rxErr = ioctl(FCAN, FCAN_GET_RX_ERR_COUNT, NULL);
```

This code reads the value of FlexCAN RX error counter.

5.7.3.37 FCAN_GET_TX_ERR_COUNT - read TX error counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase, FCAN_GET_TX_ERR_COUNT,
          NULL);
```

Arguments:

Table 5-218. FCAN_GET_TX_ERR_COUNT ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.

Description: The *FCAN_GET_TX_ERR_COUNT* ioctl command returns the 8bit value of the FlexCAN TX error counter. See the FlexCAN documentation for detailed description of error counter values.

Returns: Content of the TX error counter field of the error counter register (FC_ERR_CNTRS).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCAN_GET_TX_ERR_COUNT* ioctl command is implemented as a macro.

Example 5-197. FCAN_GET_TX_ERR_COUNT

```
UWord16 txErr;

txErr = ioctl(FCAN, FCAN_GET_TX_ERR_COUNT, NULL);
```

This code reads the value of FlexCAN TX error counter.

5.7.3.38 FCAN_GET_MB_MODULE - get pointer to Message Buffer module

Call(s):

```
FCAN_MB* ioctl(const int *pModuleBase, FCAN_GET_MB_MODULE,
          UWord16 param);
```

Arguments:

Table 5-219. FCAN_GET_MB_MODULE ioctl call arguments

pModuleBase	in	The FlexCAN module identifier. Use FCAN and FCAN2. Note that FCAN2 is valid only on 56F8367.
param	in	The Message Buffer index.

Description: The *FCAN_GET_MB_MODULE* ioctl command returns the pointer to the Message Buffer structure in the FlexCAN peripheral address space. The requested MB is specified by numerical index. The returned value can be used in consecutive ioctl calls to configure the requested Message Buffer. The Message Buffer-oriented ioctl commands are described in the following sections.

Returns: Pointer to system Message Buffer structure of type `FCAN_MB`.

Range Issues: The ioctl parameter identifies the Message Buffer index. Valid range is 0...15.

Special Issues: None.

Design/Implementation: The *FCAN_GET_MB_MODULE* ioctl command is implemented as a macro.

Example 5-198. FCAN_GET_MB_MODULE

```
FCAN_MB* pmb;

pmb = ioctl(FCAN, FCAN_GET_MB_MODULE, 5);
```

This code retrieves pointer to Message Buffer module number 5.

5.7.3.39 FCANMB_GET_ID - get message identifier

Call(s):

```
UWord32 ioctl(FCAN_MB *pMBBase, FCANMB_GET_ID, NULL);
```

Arguments:

Table 5-220. FCANMB_GET_ID ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_ID* ioctl command retrieves the CAN frame identifier of the message currently stored in Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

The returned Message Identifier is converted from a raw form of the MB structure so that it can be treated as a standard 32bit number. The most significant bit (MSB) of the return value is set for the extended 29bit identifiers. The MSB is cleared for the standard 11bit identifiers. Several examples of possible return values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567
0x80000033   ... Extended identifier 0x33
```

Returns: CAN frame identifier value. The most significant bit identifies extended (MSB=1) or standard (MSB=0) CAN identifier. You can use the FCAN_ID_EXT constant to test the MSB bit.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_ID* ioctl command is implemented as a function call. The function does not change any registers except the function return value register.

Example 5-199. FCANMB_GET_ID

```
UWord32 id;

id = ioctl(FCAN_MB5, FCANMB_GET_ID, NULL);
```

This code retrieves frame Id of MB5.

5.7.3.40 FCANMB_GET_ID_RAW - get message identifier

Call(s):

```
UWord32 ioctl(FCAN_MB *pMBBase, FCANMB_GET_ID_RAW, NULL);
```

Arguments:

Table 5-221. FCANMB_GET_ID_RAW ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_ID_RAW* ioctl command retrieves the CAN frame identifier of the message currently stored in Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

The returned value is in the raw format as it is stored in the MB registers so further conversions are needed to obtain numeric representation of the message identifier.

Returns: CAN frame identifier value in raw (registers) format.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_ID_RAW* ioctl command is implemented as a macro.

Example 5-200. FCANMB_GET_ID_RAW

```
UWord32 id;

id = ioctl(FCAN_MB5, FCANMB_GET_ID_RAW, NULL);
```

This code retrieves raw frame Id of MB5.

5.7.3.41 FCANMB_GET_LEN - get data length

Call(s):

```
UWord16 ioctl(FCAN_MB* pMBBase, FCANMB_GET_LEN, NULL);
```

Arguments:

Table 5-222. FCANMB_GET_LEN ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_LEN* ioctl command retrieves the length of the frame data part currently stored in Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

Returns: Length of the message stored in message buffer (number 0...8)

Range Issues: The parameter passed to this ioctl command must be the one obtained using *FCAN_GET_MB_MODULE* command.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_LEN* ioctl command is implemented as a macro.

Example 5-201. FCANMB_GET_LEN

```
UWord16 len;

len = ioctl(FCAN_MB5, FCANMB_GET_LEN, NULL);
```

This code retrieves the length of the message in MB5.

5.7.3.42 FCANMB_GET_DATAPTR - get pointer to frame data

Call(s):

```
UWord16* ioctl(FCAN_MB* pMBBase, FCANMB_GET_DATAPTR, NULL);
```

Arguments:

Table 5-223. FCANMB_GET_DATAPTR ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_DATAPTR* ioctl command returns the pointer to data part of the message currently stored in Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

The returned pointer points to four 16-bit data words in which up to eight message bytes are stored in big endian format. Use the *FCANMB_GET_LEN* command to determine the number of valid data bytes in the message.

Returns: Pointer to four data words containing up to eight data bytes stored in big endian format.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_DATAPTR* ioctl command is implemented as a macro.

Example 5-202. FCANMB_GET_DATAPTR

```
UWord16 len;
UWord16* pdata

len = ioctl(FCAN_MB5, FCANMB_GET_LEN, NULL);
pdata = ioctl(FCAN_MB5, FCANMB_GET_DATAPTR, NULL);
```

This code retrieves the length and pointer to data part of the frame in MB5.

5.7.3.43 FCANMB_GET_CODE - get message buffer status code

Call(s):

```
UWord16 ioctl(FCAN_MB* pMBBase, FCANMB_GET_CODE, NULL);
```

Arguments:

Table 5-224. FCANMB_GET_CODE ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_CODE* ioctl command retrieves current status code of the Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

The returned value can be compared with the set of predefined code values:

```
FCAN_MB_CODE_RXVOID       buffer void after received data read-out
FCAN_MB_CODE_RXEMPTY      listening active and empty
FCAN_MB_CODE_RXFULL       filled with received data
FCAN_MB_CODE_RXOVERRUN    receiver overrun (MB contains last message data)
FCAN_MB_CODE_RXBUSY       just being filled with new data
FCAN_MB_CODE_TXVOID       buffer void before new TX data can be copied into it
FCAN_MB_CODE_TXONCE       queued for transmission, once
FCAN_MB_CODE_TXRALWAYS    transmit response to RTR frame, always
```

Be aware that in some cases, reading of MB code locks the Message Buffer for CPU access. Such a lock then prevents the FlexCAN module to fill a new received data into MB. Use the *FCAN_UNLOCK_ALL_MB* ioctl command to unlock the locked Message Buffer.

Returns: A message buffer code value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_CODE* ioctl command is implemented as a macro.

Example 5-203. FCANMB_GET_CODE

```
UWord16 code = ioctl(FCAN_MB5, FCANMB_GET_CODE, NULL);

if(code == FCAN_MB_CODE_RXFULL || code == FCAN_MB_CODE_RXOVERRUN)
{
      ... // read out the data
      ioctl(FCAN, FCAN_UNLOCK_ALL_MB, NULL); // unlock MB
}
```

This code tests the status code of the MB5.

5.7.3.44 FCANMB_GET_TIMESTAMP - get message time stamp

Call(s):

```
UWord16 ioctl(FCAN_MB* pMBBase, FCANMB_GET_TIMESTAMP, NULL);
```

Arguments:

Table 5-225. FCANMB_GET_TIMESTAMP ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_TIMESTAMP* ioctl command retrieves the time stamp of the message in the Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

For standard-identifier messages, the full 16bit time stamp value is available and returned by this command. For extended-identifier messages, only the upper byte of the time stamp value is available in the Message Buffer system structure so the lower byte of the returned value is zeroed.

Because the time stamp value is stored in the same physical register as the MB status code value, a reading of the time stamp value may lock the Message Buffer for CPU access (see *FCANMB_GET_CODE*). Such a lock then prevents the FlexCAN module to fill a new received data into MB. Use the *FCAN_UNLOCK_ALL_MB* ioctl command to unlock the locked Message Buffer.

Returns: A time stamp value of the message in the Message Buffer.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_TIMESTAMP* ioctl command is implemented as a macro.

Example 5-204. FCANMB_GET_TIMESTAMP

```
UWord16 ts;

ts = ioctl(FCAN_MB5, FCANMB_GET_TIMESTAMP, NULL);
```

This code retrieves the time stamp of the message received in MB5.

5.7.3.45 FCANMB_GET_TIMESTAMP8 - get message time stamp

Call(s):

```
UWord16 ioctl(FCAN_MB* pMBBase, FCANMB_GET_TIMESTAMP8, NULL);
```

Arguments:

Table 5-226. FCANMB_GET_TIMESTAMP8 ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367

Description: The *FCANMB_GET_TIMESTAMP8* ioctl command retrieves the time stamp of the message in the Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

Unlike the *FCANMB_GET_TIMESTAMP* command, this command retrieves always only the upper byte of the time stamp value, which is available for both standard and extended identifier messages. In comparison with the *FCANMB_GET_TIMESTAMP* this command saves the execution time of the code which tests the type of the message stored in the MB.

Because the time stamp value is stored in the same physical register as the MB status code value, the reading of the time stamp value may lock the Message Buffer for CPU access (see *FCANMB_GET_CODE*). Such a lock then prevents the FlexCAN module to fill a new received data into MB. Use the *FCAN_UNLOCK_ALL_MB* ioctl command to unlock the locked Message Buffer.

Returns: A time stamp value of the message in the Message Buffer.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_GET_TIMESTAMP8* ioctl command is implemented as a macro.

Example 5-205. FCANMB_GET_TIMESTAMP8

```
UWord16 ts;

ts = ioctl(FCAN_MB5, FCANMB_GET_TIMESTAMP8, NULL);
```

This code retrieves the time stamp of the message received in MB5.

5.7.3.46 FCANMB_SET_ID - set message identifier

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_ID, UWord32 param);
```

Arguments:

Table 5-227. FCANMB_SET_ID ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	32bit Identifier value (constant). The most significant bit, when set, identifies the extended message identifier. Use an OR-combination of the ID numeric value and the following constants: FCAN_ID_EXT ... when passing extended identifier FCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *FCANMB_SET_ID* ioctl command assigns the CAN message identifier to the specified Message Buffer, and clears or sets the Remote Transmit Request flag in the MB. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

The identifier value passed to this ioctl command should contain information about whether it is specified in standard 11bit format or extended 29bit format. The most significant bit (MSB) is reserved for this purpose and you can use the predefined FCAN_ID_EXT constant OR-ed with the ID value being passed to the ioctl. Several examples of composing the identifier and mask values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567 (= 0x1234567 | FCAN_ID_EXT)
0x80000033   ... Extended identifier 0x33 (= 0x33 | FCAN_ID_EXT)
```

By default, this command clears the RTR flag, which is stored in ID_LOW or ID_HIGH Message Buffer registers - depending on the message type. Use the bit30 of the ID passed to this command to set the RTR bit in appropriate register. The bit30 is defined as FCAN_ID_RTR constant in *fcan.h*. You can also use the *FCANMB_SET_RTR* command to set or clear the RTR bit after the frame ID is already stored in the Message Buffer.

Returns: None.

Range Issues: The parameter contains the message id for which the following range apply: Standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. Extended identifiers (MSB=1) are limited to 29 bits and the valid range is 0x8000000... 0x9fffffff.

Special Issues: This command uses *FCAN_Id2Idr()* macro to convert the passed value into the raw format suitable for writing into ID_LOW and ID_HIGH registers. The conversion process is not trivial and may produce quite lengthy code when passing a UWord32 variable as the parameter. Use this command for passing constant values only and use the *FCANMB_SET_ID_V* command, which is operational equivalent implemented as a function call.

Design/Implementation: The *FCANMB_SET_ID* ioctl command is implemented as a macro.

Example 5-206. FCANMB_SET_ID

```
ioctl(FCAN_MB5, FCANMB_SET_ID, 0x123 | FCAN_ID_EXT | FCAN_ID_RTR);
```

This code sets the message identifier of the MB5 to Extended ID 0x123. It also sets the RTR bit.

5.7.3.47 FCANMB_SET_ID_V - set message identifier

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_ID_V, UWord32 param);
```

Arguments:

Table 5-228. FCANMB_SET_ID_V ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	32bit Identifier value (constant). The most significant bit, when set, identifies the extended message identifier. Use an OR-combination of the ID numeric value and the following constants: FCAN_ID_EXT ... when passing extended identifier FCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *FCANMB_SET_ID_V* ioctl command assigns the CAN message identifier to the specified Message Buffer. This command operates the same way as the *FCANMB_SET_ID*, except that *FCANMB_SET_ID_V* command is implemented as function call and is thus better suitable for passing the 32bit identifier stored in the caller's variable.

Returns: None.

Range Issues: The first parameter passed to this ioctl command must be the value obtained using *FCAN_GET_MB_MODULE* command. The second parameter contains the message id for which the following range apply: Standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. Extended identifiers (MSB=1) are limited to 29 bits and the valid range is 0x8000000... 0x9fffffff.

Special Issues: None.

Design/Implementation: The *FCANMB_SET_ID_V* ioctl command is implemented as a function call. This function call saves the contents of the CPU registers, so it can be called from interrupt service routines.

Example 5-207. FCANMB_SET_ID_V

```
UWord32 id = 0x123 | FCAN_ID_EXT;

ioctl(FCAN_MB5, FCANMB_SET_ID_V, id);
```

This code sets the message identifier of the MB5 to Extended ID 0x123.

5.7.3.48 FCANMB_SET_ID_RAW - set message identifier

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_ID_RAW, UWord32 param);
```

Arguments:

Table 5-229. FCANMB_SET_ID_RAW ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	32bit Identifier value (constant). The most significant bit, when set, identifies the extended message identifier. Use an OR-combination of the ID numeric value and the following constants: FCAN_ID_EXT ... when passing extended identifier FCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *FCANMB_SET_ID_RAW* ioctl command assigns the CAN message identifier to the specified Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

This command behaves similarly as the *FCANMB_SET_ID*, except that passed identifier value must already be in the raw format suitable for writing into high and low registers of the Message Buffer structure.

Returns: None.

Range Issues: See the FlexCAN module documentation for the details about Message Buffer identifier registers.

Special Issues: None.

Design/Implementation: The *FCANMB_SET_ID_RAW* ioctl command is implemented as a macro.

Example 5-208. FCANMB_SET_ID_RAW

```
ioctl(FCAN_MB5, FCANMB_SET_ID_RAW, 0x11 << 21);
```

This code sets the message identifier of the MB5 to 0x11.

5.7.3.49 FCANMB_SET_RTR - set Remote Transmit Request flag

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_RTR, UWord16 param);
```

Arguments:

Table 5-230. FCANMB_SET_RTR ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	Parameter to select the desired action. Use one of these predefined constants: FCAN_ON - set RTR FCAN_OFF - clear RTR

Description: The *FCANMB_SET_RTR* ioctl command sets or clears the Remote Transmit Request (RTR) bit in the message identifier value of the specified Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

As an alternative to this command, you can set the RTR bit directly when setting the Message Buffer ID value using the *FCANMB_SET_ID* or *FCANMB_SET_ID_V* commands. The RTR bit is also directly accessed when writing a raw ID value using the *FCAN_SET_ID_RAW* command.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *FCANMB_SET_RTR* ioctl command is implemented as a macro.

Example 5-209. FCANMB_SET_RTR

```
ioctl(FCAN_MB5, FCANMB_SET_LEN, 0);
ioctl(FCAN_MB5, FCANMB_SET_ID, 0x11);
ioctl(FCAN_MB5, FCANMB_SET_RTR, FCAN_ON);
```

This code prepares the transmission of the Remote Transmit Request of the id 0x11 from the MB5.

5.7.3.50 FCANMB_SET_LEN - set message length

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_LEN, UWord16 param);
```

Arguments:

Table 5-231. FCANMB_SET_LEN ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	The length of the message to be set to the Message Buffer.

Description: The *FCANMB_SET_LEN* ioctl command sets the message length field of the specified Message Buffer structure. Typically the length is set before the message is to be transmitted onto the CAN bus. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command.

Returns: None.

Range Issues: The parameter must be in the range from 0 to 8.

Special Issues: This command is implemented as read-modify-write of the Message Buffer control register, which has the side effect of locking the MB. Mostly, this is not an issue because the *FCANMB_SET_LEN* is used before *transmitting* the MB where the locking mechanism does not apply. On the other side, locking of one MB causes release of the lock of another MB which might cause a problem when not taken into consideration.

Design/Implementation: The *FCANMB_SET_LEN* ioctl command is implemented as a macro.

Example 5-210. FCANMB_SET_LEN

```
ioctl(FCAN_MB5, FCANMB_SET_LEN, 4);
```

This code sets the message length of the MB5 to four.

5.7.3.51 FCANMB_SET_CODE - set message buffer action code

Call(s):

```
void ioctl(FCAN_MB* pMBBase, FCANMB_SET_CODE, UWord16 param);
```

Arguments:

Table 5-232. FCANMB_SET_CODE ioctl call arguments

pMBBase	in	The FlexCAN Message Buffer module identifier. Use the value returned from FCAN_GET_MB_MODULE command or one of the predefined MB module identifiers: FCAN_MB0, FCAN_MB1, ... FCAN_MB15 for FlexCAN module, or FCAN2_MB0, FCAN2_MB1, ... FCAN2_MB15 for FlexCAN2 module on 56F8367
param	in	The action code to be set to the Message Buffer. Use predefined constants only (see list below).

Description: The *FCANMB_SET_CODE* ioctl command sets the action code of the specified Message Buffer. The requested MB is specified as the first argument of the ioctl call by the pointer to its system structure. You can use one of the predefined MB module identifiers or use the value dynamically retrieved using *FCAN_GET_MB_MODULE* command

Using this commands and selecting the proper action code, the Messages Buffer can be configured for various transmission or reception modes. The list of action codes which can be assigned to a Message Buffer follows:

```
FCAN_MB_CODE_RXVOID       buffer void
FCAN_MB_CODE_RXEMPTY      listening active
FCAN_MB_CODE_TXVOID       buffer void before TX
FCAN_MB_CODE_TXONCE       queued for transmission, once
FCAN_MB_CODE_TXRALWAYS    auto-transmit response to RTR frame, always
```

Returns: None.

Range Issues: None.

Special Issues: This command is implemented as read-modify-write of the Message Buffer control register, which has the side effect of locking the MB. Use the *FCAN_UNLOCK_ALL_MB* command when configuring the receiving MBs using this command (transmission MBs are not subject to locking).

Design/Implementation: The *FCANMB_SET_CODE* ioctl command is implemented as a macro.

Example 5-211. FCANMB_SET_CODE

```
UWord16* pd;
ioctl(FCAN_MB5, FCANMB_SET_CODE, FCAN_MB_CODE_TXVOID);
pd = ioctl(FCAN_MB5, FCANMB_GET_DATAPTR, NULL);
pd[0] = 0x1234;
ioctl(FCAN_MB5, FCANMB_SET_LEN, 2);
ioctl(FCAN_MB5, FCANMB_SET_ID, 0x12 | FCAN_ID_EXT);
ioctl(FCAN_MB5, FCANMB_SET_CODE, FCAN_MB_CODE_TXONCE);
```

This code transmits a two-byte message with extended identifier 0x12 from MB5.

5.7.4 FlexCAN Driver Sample Application

The FlexCAN driver application is designed for Freescale/Motorola EVM's and is intended to illustrate the usage of this driver by a real example.

The application shows a static initialization of the FlexCAN module and its run-time use. The static initialization is performed by the *FCAN_INIT* command. See the items in *appconfig.h*, which are written into the FlexCAN registers.

The sample application demonstrated the following features of the FlexCAN module and FlexCAN low-level driver

1. The application uses MB14 to listen and look for CAN frames of standard ID 0x11. When such is found and its data part is at least two bytes, the echo frame is sent from MB0. Echo frame is assigned extended ID 0x1111 and is two bytes long. The two data bytes are copied from the received message. The red LED toggles each time the message is sent.
2. In the main loop, the MB5 is used to periodically transmit data (Ext ID: 0x333333)
3. The MB 10 is used to demonstrate RTR automatic responses (ID: 0x66)

The application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\fcan_demo* directory and consists of the application project *fcan_demo.mcp*, source code for the application *main.c* and configuration file *appconfig.h*.

Example 5-212. FlexCAN driver sample application appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*   File generated by Graphical Configuration Tool
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 60.000 MHz
    VCO frequency: 240.000 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.389 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enable , Wait enable
         Ext. bus driven when inactive: Disable
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
```

```
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output: CLKO select: sys_clk (OCCS) , CLKOUT mode: Tristated
    SEMI - CS0: Base address: 0x0, Blocksize: 128K , BYTE_EN: Both bytes enable , R/W:
Read / Write
                PS/DS select: PS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS1: Base address: 0x0, Blocksize: 128K , BYTE_EN: Lower byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS2: Base address: 0x0, Blocksize: 128K , BYTE_EN: Upper byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS3: Base address: 0x0, Blocksize: 32K , BYTE_EN: Disable , R/W: Disable
                PS/DS select: Disable , Write Wait States: 23, Read Wait States: 23
.*/
#define SEMI_CSBAR0_INIT                  0x0005
#define SEMI_CSBAR1_INIT                  0x0005
#define SEMI_CSBAR2_INIT                  0x0005
#define SEMI_CSOR0_INIT                   0x1FC3
#define SEMI_CSOR1_INIT                   0x1BA3
#define SEMI_CSOR2_INIT                   0x1DA3
#define SIM_GPS_INIT                      0x0000

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                    0x0000
#define INT_VECTOR_ADDR_26                CAN_BusOffISR
#define INT_PRIORITY_LEVEL_26             INTC_LEVEL0
#define INT_VECTOR_ADDR_27                CAN_ErrorISR
#define INT_PRIORITY_LEVEL_27             INTC_LEVEL0
#define INT_VECTOR_ADDR_28                CAN_WakeUpISR
#define INT_PRIORITY_LEVEL_28             INTC_LEVEL0
#define INT_VECTOR_ADDR_29                CAN_MbISR
#define INT_PRIORITY_LEVEL_29             INTC_LEVEL0

/*.
    FCAN Configuration
-----------------------------------------
    Baudrate: 500 kBaud
    Keep in low-power STOP mode: No
    Self Wake Up: Disabled
    Halt/Freeze sensitivity: Disabled
    Keep in Halt/Freeze: No
    Auto power save: Disabled
    Timer Synchronization mode: Disabled
    Loopback mode: Disabled
    Listen Only mode: Disabled
    Sampling: One sample per bit
    Transmission: Lowest ID transmitted first
    Prescaler Divide Factor: 12
    Propagation Segment (PROP_SEG): 1
    Phase Buffer Segment 1 (PHASE_SEG1): 4
    Phase Buffer Segment 2 (PHASE_SEG2): 4
    Re-Synchronization Jump Width (RJW): 1
    Interrupts: Wake up: Enable
                Error  : Enable
                Bus Off: Enable
    MB interrupts: MB0 : No , MB1 : No , MB2 : No , MB3 : No
                   MB4 : No , MB5 : No , MB6 : No , MB7 : No
                   MB8 : No , MB9 : No , MB10: No , MB11: No
```

```
                    MB12: No , MB13: No , MB14: No , MB15 : No
.*/
#define FCAN_MCR_INIT                       0x0400
#define FCAN_CTL0_INIT                      0xC000
#define FCAN_CTL1_INIT                      0x0B1B
#define FCAN_RXGMASKL_INIT                  0xFFFE
#define FCAN_RXGMASKH_INIT                  0xFFEF
#define FCAN_RX14MASKL_INIT                 0xFFFE
#define FCAN_RX14MASKH_INIT                 0xFFEF
#define FCAN_RX15MASKL_INIT                 0xFFFE
#define FCAN_RX15MASKH_INIT                 0xFFEF


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-213. FlexCAN driver sample application main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION:  FlexCAN low-level driver demo application
*
*    1. This application uses MB14 to listen and look for CAN frames
*       of std ID 0x11. When such is found and its data part is at least
*       two bytes, the echo frame is sent from MB0. Echo frame is assigned
*       ext ID 0x1111 and is two bytes long. The two data bytes are copied
*       from the received message.
*       The green LED toggles each time the message is sent.
*
*    2. In the main loop, the MB5 is used to periodically transmit data
*       (Ext ID: 0x333333)
*
*    3. The MB 10 is used to demonstrate RTR automatic responses (ID: 0x66)
*
*
* TARGET: MC56F8346 device
*
*****************************************************************************/

#include "qs.h"

#include "intc.h"
#include "gpio.h"
#include "occs.h"
#include "fcan.h"


/*****************************************************************************
             Interrupt Service Routines
*****************************************************************************/

#pragma interrupt on

void CAN_BusOffISR()
{
             /* acknowledge the interrupt */
             ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
             ioctl(FCAN, FCAN_CLEAR_BOFF_INT, NULL);
}

void CAN_WakeUpISR()
{
             /* acknowledge the interrupt */
             ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
             ioctl(FCAN, FCAN_CLEAR_WAKE_INT, NULL);
}

void CAN_ErrorISR()
{
             /* acknowledge the interrupt */
             ioctl(FCAN, FCAN_READ_ERR_AND_STATUS, NULL);
             ioctl(FCAN, FCAN_CLEAR_ERR_INT, NULL);
}
```

```
void CAN_MbISR()
{
            register UWord16 src;

            /* get interrupt sources */
            src = ioctl(FCAN, FCAN_READ_MBINT_FLAGS, NULL);

            /* MB14 needs to service ? */
            if(src & FCAN_MBINT_14)
            {
                  /* read MB status code (= lock MB) */
                  register UWord16 code = ioctl(FCAN_MB14, FCANMB_GET_CODE, NULL);

                  /* data received ? */
                  if(code == FCAN_MB_CODE_RXFULL || code == FCAN_MB_CODE_RXOVERRUN)
                  {
                        /* at least two bytes must be received ! */
                        if(ioctl(FCAN_MB14, FCANMB_GET_LEN, NULL) >= 2)
                        {
                              /* yes, we will send the echo back */
                              UWord16 d = * ioctl(FCAN_MB14, FCANMB_GET_DATAPTR,
                                                NULL);

                              /* prepare MB0 for transmission, (note that here we
                                 are releasing
                                 MB14 lock as this command is implemented as
                                 READ-modify-write) */
                              ioctl(FCAN_MB0, FCANMB_SET_CODE,
                                    FCAN_MB_CODE_TXVOID);

                              /* copy first two bytes of data */
                              * ioctl(FCAN_MB0, FCANMB_GET_DATAPTR, NULL) = d;

                              /* set transmission ID */
                              ioctl(FCAN_MB0, FCANMB_SET_ID, 0x1111 |
                                    FCAN_ID_EXT);

                              /* set transmission length */
                              ioctl(FCAN_MB0, FCANMB_SET_LEN, 2);

                              /* transmit  */
                              ioctl(FCAN_MB0, FCANMB_SET_CODE,
                                    FCAN_MB_CODE_TXONCE);

                              /* toggle LED */
                              ioctl(GPIO_C, GPIO_TOGGLE_PIN, BIT_2);
                        }
                  }

                  /* unlock MB */
                  ioctl(FCAN, FCAN_UNLOCK_ALL_MB, NULL);

                  /* note that MB remains configured as RX with the original ID  */

            }

            /* acknowledge any MB interrupt */
            ioctl(FCAN, FCAN_CLEAR_MBINT_FLAGS, 0xffff);
}

#pragma interrupt off
```

```
/*******************************************************************************
            main
*******************************************************************************/

int main(void)
{
    volatile UWord16* pdata;
    UWord16 i;

    /* initialize interrupts */
    ioctl(INTC, INTC_INIT, NULL);
    archEnableInt();

    /*initialization of OCCS module based on config items from appconfig.h */
    ioctl(OCCS, OCCS_INIT, NULL);

    /* initialize green LED PC5 */
    ioctl(GPIO_C, GPIO_SETAS_GPIO, BIT_2);
    ioctl(GPIO_C, GPIO_SETAS_OUTPUT, BIT_2);

    /* initialize FlexCAN (speed, masks etc.) */
    ioctl(FCAN, FCAN_INIT, NULL);

    /* enable MB14 interrupt */
    ioctl(FCAN, FCAN_MBINT_ENABLE, FCAN_MBINT_14);

    /* start listening on MB14 id 0x11 */
    ioctl(FCAN_MB14, FCANMB_SET_CODE, FCAN_MB_CODE_RXVOID);
    ioctl(FCAN_MB14, FCANMB_SET_ID, 0x11);
    ioctl(FCAN_MB14, FCANMB_SET_CODE, FCAN_MB_CODE_RXEMPTY);

    /* configure MB 10 for RTR auto-transmission on ID 0x66 */
    ioctl(FCAN_MB10, FCANMB_SET_CODE, FCAN_MB_CODE_TXVOID);
    ioctl(FCAN_MB10, FCANMB_SET_ID, 0x66);
    ioctl(FCAN_MB10, FCANMB_SET_LEN, 4);
    pdata = ioctl(FCAN_MB10, FCANMB_GET_DATAPTR, NULL);
    pdata[0] = 0x1122;// big endian encoding
    pdata[1] = 0x3344;
    ioctl(FCAN_MB10, FCANMB_SET_CODE, FCAN_MB_CODE_TXRALWAYS);

    /* configure MB 5 for periodical transmission in the endless loop bellow */
    ioctl(FCAN_MB5, FCANMB_SET_CODE, FCAN_MB_CODE_TXVOID);
    ioctl(FCAN_MB5, FCANMB_SET_ID, 0x333333 | FCAN_ID_EXT);
    ioctl(FCAN_MB5, FCANMB_SET_LEN, 5);
    pdata = ioctl(FCAN_MB5, FCANMB_GET_DATAPTR, NULL);
    pdata[0] = 0xAABB;
    pdata[1] = 0xCCDD;
    pdata[2] = 0xEE00;

    /* endless loop */
    while(1)
    {
        /* transmit MB 5 (if ready) */
        if(ioctl(FCAN_MB5, FCANMB_GET_CODE, NULL) == FCAN_MB_CODE_TXVOID)
           ioctl(FCAN_MB5, FCANMB_SET_CODE, FCAN_MB_CODE_TXONCE);

        /* wait a while */
        for(i=0; i<100; i++)
           archDelay(0xffff);
    }
}
```

5.8 GPIO Driver

This section describes the API for the MC56F83xx and MC56F80xx General Purpose Input/Output - GPIO. The functionality of the GPIO itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.8.1 Introduction

The 56F8xxx can have up to six General Purpose I/O ports GPIO Port A, Port B, Port C, Port D, Port E and Port F. GPIO pins are either dedicated or shared with another peripheral module or modules. Shared pins may be assigned to one of peripheral modules or set as a GPIO input or output. Each port can generate an interrupt. Note, that some pins are shared and some are dedicated to GPIO.

5.8.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.8.3.

Table 5-233. GPIO Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F832x	MC56F83xx	MC56F8367
Port A (PortA_BASE)	0xF180	0xF100	0xF150	0xF2E0	0xF2E0	0xF2E0
Port B (PortB_BASE)	0xF1A0	0xF110	0xF160	0xF300	0xF300	0xF300
Port C (PortC_BASE)	0xF1C0	0xF120	0xF170	0xF310	0xF310	0xF310
Port D (PortD_BASE)	0xF1E0	0xF130	0xF180/A	N/A	0xF320	0xF320
Port E (PortE_BASE)	0xF200	N/A	N/A	N/A	0xF330	0xF330
Port F (PortF_BASE)	0xF220	N/A	N/A	N/A	0xF340	0xF340

5.8.2.1 API Definition

The following header files are needed in order to use the GPIO device driver:

Required Header File(s):

```
#include "qs.h"
#include "gpio.h"
```

Public Data Structure(s):

N/A

5.8.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static GPIO port configuration by the driver initialization routine. Symbol *x* in the table should be replaced with the name of the GPIO port e.g. *GPIO_A_PUR*. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-246 for more details.

Table 5-234. GPIO Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
GPIO_*x*_PUR_INIT	UWord16	Represents contents of the GPIO Pull-Up Enable Register.
GPIO_*x*_DDR_INIT	UWord16	Represents contents of the GPIO Data Direction Register.
GPIO_*x*_PER_INIT	UWord16	Represents contents of the GPIO Peripheral Enable Register.
GPIO_*x*_IENR_INIT	UWord16	Represents contents of the GPIO Interrupt Enable Register.
GPIO_*x*_IPOLR_INIT	UWord16	Represents contents of the GPIO Interrupt Polarity Register.
MC56F83xx, MC56F801x, MC56F802x/3x only: GPIO_*x*_PPMODE_INIT	UWord16	Represents contents of the GPIO Push-Pull Mode Register.
MC56F80xx only: GPIO_*x*_DRIVE_INIT	UWord16	Represents contents of the GPIO Drive Strength Register.
MC56F800x only: GPIO_*x*_IFE_INIT	UWord16	Represents contents of the GPIO Input Filter Register.
MC56F800x only: GPIO_*x*_SLEW_INIT	UWord16	Represents contents of the GPIO Slew Rate Register.

5.8.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
void ioctl(const int *pModuleBase, void Cmd, void *pParam);
UWord16 ioctl(const int *pModuleBase, void Cmd, void *pParam);
```

Description: The *ioctl* call "changes" the GPIO device modes or accesses the GPIO register(s).

Arguments:

Table 5-235. GPIO Driver Arguments - ioctl

pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F.
Cmd	in	Commands found in *gpio.h* which are used to modify the GPIO module status and control registers. See Table 5-236.
pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-236. ioctl commands

Cmd	pParam	Return	Description
GPIO_INIT	None	None	Initializes selected GPIO port by data from configuration file (app-config.h).

Table 5-236. ioctl commands (Continued)

Cmd	pParam	Return	Description
GPIO_INIT_ALL	None	None	Initializes all GPIO ports by data from configuration file (appconfig.h).
GPIO_SETAS_GPIO	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected pins as GPIO.
GPIO_SETAS_PERIPHERAL	BIT_0 | BIT_1 | ... | BIT_15	None	Assigns the selected pins to a peripheral.
GPIO_SETAS_INPUT	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected pins as input pins.
GPIO_SETAS_OUTPUT	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected pins as output pins.
GPIO_INT_DISABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Disables interrupt request generated by GPIO.
GPIO_INT_ENABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Enables interrupt request generated by GPIO.
GPIO_PULLUP_DISABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Disables pull-up when the selected pins are configured as GPIO inputs.
GPIO_PULLUP_ENABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Enables pull-up when the selected pins are configured as GPIO inputs.
MC56F83xx ,MC56F801x, MC56F802x/3x only: GPIO_CLEAR_SW_INT_PENDING	BIT_0 | BIT_1 | ... | BIT_15	None	Disables interrupt request generated by software (only for software testing).
MC56F83xx ,MC56F801x, MC56F802x/3x only: GPIO_SW_INT_ASSERT	BIT_0 | BIT_1 | ... | BIT_15	None	Enables interrupt request generated by software (only for software testing).
GPIO_INT_DETECTION_ACTIVE_HIGH	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected GPIO pin to be active high.
GPIO_INT_DETECTION_ACTIVE_LOW	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected GPIO pin to be active low.
GPIO_CLEAR_INT_PENDING	BIT_0 | BIT_1 | ... | BIT_15	None	Clears interrupt request flags.
GPIO_SET_PIN	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the selected GPIO pins of the port.
GPIO_CLEAR_PIN	BIT_0 | BIT_1 | ... | BIT_15	None	Clears the selected GPIO pins of the port.
GPIO_TOGGLE_PIN	BIT_0 | BIT_1 | ... | BIT_15	None	Toggles the selected GPIO pins of the port.

Table 5-236. ioctl commands (Continued)

Cmd	pParam	Return	Description
GPIO_READ_DATA	None	UWord16	Reads from the selected GPIO port.
GPIO_WRITE_DATA	UWord16	None	Writes to the selected GPIO port.
GPIO_READ_INT_PENDING_REG	None	UWord16	Reads GPIO Interrupt Pending Register.
GPIO_GET_INT_PENDING_FLAG	BIT_0 | BIT_1 | ... | BIT_15	UWord16	Gets the selected GPIO interrupt pending flag(s).
GPIO_TEST_INT_PENDING	BIT_0 | BIT_1 | ... | BIT_15	UWord16	Tests the selected GPIO interrupt pending flag(s).
MC56F83xx ,MC56F801x, MC56F802x/3x only: GPIO_SETAS_PUSHPULL	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the output driver of the selected GPIO pins to push-pull mode.
MC56F83xx ,MC56F801x, MC56F802x/3x only: GPIO_SETAS_OPENDRAIN	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the output driver of the selected GPIO pins to open drain mode.
GPIO_READ_RAW_DATA	NULL	UWord16	Reads the logic value of each GPIO pin on the selected port.
MC56F80xx only: GPIO_SET_HIGH_DRIVE_STRENGTH	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the high strength of the selected output driver.
MC56F80xx only: GPIO_SET_LOW_DRIVE_STRENGTH	BIT_0 | BIT_1 | ... | BIT_15	None	Sets the low strength of the selected output driver.
MC56F800x only: GPIO_SET_LOW_PASS_FILTER_ENABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Enables input low pass filter
MC56F800x only: GPIO_SET_LOW_PASS_FILTER_DISABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Disables input low pass filter
MC56F800x only: GPIO_SET_LOW_SLEW_RATE_ENABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Enables slew rate on selected GPIO pins
MC56F800x only: GPIO_SET_LOW_SLEW_RATE_DISABLE	BIT_0 | BIT_1 | ... | BIT_15	None	Disables slew rate on selected GPIO pins

5.8.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the ioctl commands usage.

5.8.3.1 GPIO_INIT - initialize GPIO port

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_INIT, NULL);
```

Arguments:

Table 5-237. GPIO_INIT ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.

Description: The *GPIO_INIT* ioctl command is used to initialize the selected GPIO module. Initialization consists of writing the initialization values to the following GPIO registers: GPIO_Pull-Up Enable Register (GPIO_x_PUR), Data Direction Register (GPIO_x_DDR), Peripheral Enable Register (GPIO_x_PER), Interrupt Enable Register (GPIO_x_IENR), Interrupt Polarity Register (GPIO_x_IPOLR) and Push-Pull Mode Register (GPIO_x_PPMODE). The Drive Strength Control Register (GPIO_x_DRIVE) is initialized on 56F801x also. Initialization values (configuration items) for each register are taken from *appconfig.h*, where each configuration item corresponds to one GPIO register. If no initialization value is defined in *appconfig.h*, then no initialization is performed. It is not necessary to define the initialization values for all GPIO registers in *appconfig.h*, but only for those which are needed. For reference on symbols of configuration items, see Section 5.8.2.2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_INIT* ioctl command is implemented as a function call.

Example 5-214. GPIO_INIT

```
ioctl( GPIO_B, GPIO_INIT, NULL);
```

This code initializes the GPIO Port B.

5.8.3.2 GPIO_INIT_ALL - initialize all GPIO port

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_INIT_ALL, NULL);
```

Arguments:

Table 5-238. GPIO_INIT_ALL ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO.

Description: The *GPIO_INIT_ALL* ioctl command is used to initialize all GPIO modules. Initialization consists of calling the initialization ioctl commands GPIO_INIT for all GPIO ports - GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E and GPIO_F. Initialization values (configuration items) for each GPIO module are taken from *appconfig.h*. If no initialization value is defined in *appconfig.h*, then no initialization is performed. It is not necessary to define the initialization values for all GPIO registers in *appconfig.h*, but only for those which are needed.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_INIT_ALL* ioctl command is implemented as a function call.

Example 5-215. GPIO_INIT_ALL

```
ioctl( GPIO, GPIO_INIT_ALL, NULL);
```

This code initializes all GPIO Ports.

5.8.3.3 GPIO_SETAS_GPIO - set the selected pins as GPIO pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_GPIO,
           UWord16 param);
```

Arguments:

Table 5-239. GPIO_SETAS_GPIO ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_GPIO* ioctl command sets the selected GPIO pins of the GPIO port as GPIO pins by modifying content of the Peripheral Enable Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_SETAS_GPIO* ioctl command is implemented as a macro.

Example 5-216. GPIO_SETAS_GPIO

```
ioctl( GPIO_B, GPIO_SETAS_GPIO, BIT_0 | BIT_1 | BIT_2);
```

This code sets pins 0, 1 and 2 on Port B as GPIO pins.

5.8.3.4 GPIO_SETAS_PERIPHERAL - assign the selected pins to a peripheral

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_PERIPHERAL,
           UWord16 param);
```

Arguments:

Table 5-240. GPIO_SETAS_PERIPHERAL ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_PERIPHERAL* ioctl command assigns the selected GPIO pins of the GPIO to a peripheral by modifying the content of the Peripheral Enable Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_SETAS_PERIPHERAL* ioctl command is implemented as a macro.

Example 5-217. GPIO_SETAS_PERIPHERAL

```
ioctl( GPIO_B, GPIO_SETAS_PERIPHERAL, BIT_0 | BIT_1 | BIT_2);
```

This code assigns pins 0, 1 and 2 on Port B to a peripheral.

5.8.3.5 GPIO_SETAS_INPUT - set the selected pins as input pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_INPUT,
           UWord16 param);
```

Arguments:

Table 5-241. GPIO_SETAS_INPUT ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_INPUT* ioctl command sets the selected GPIO pins as an input pins by modifying the content of the Data Direction Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as a GPIO pins (See Section 5.8.3.3).

Design/Implementation: The *GPIO_SETAS_INPUT* ioctl command is implemented as a macro.

Example 5-218. GPIO_SETAS_INPUT

```
ioctl( GPIO_B, GPIO_SETAS_INPUT, BIT_0 | BIT_1 | BIT_2);
```

This code sets pins 0, 1 and 2 on Port B as input pins.

5.8.3.6 GPIO_SETAS_OUTPUT - set the selected pins as output pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_OUTPUT,
           UWord16 param);
```

Arguments:

Table 5-242. GPIO_SETAS_OUTPUT ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_OUTPUT* ioctl command sets the selected GPIO pins as an output pins by modifying the content of the Data Direction Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as a GPIO pins (See Section 5.8.3.3).

Design/Implementation: The *GPIO_SETAS_OUTPUT* ioctl command is implemented as a macro.

Example 5-219. GPIO_SETAS_OUTPUT

```
ioctl( GPIO_B, GPIO_SETAS_OUTPUT, BIT_0 | BIT_1 | BIT_2);
```

This code sets pins 0, 1 and 2 on Port B as output pins.

5.8.3.7 GPIO_INT_DISABLE - disable interrupt request generated by GPIO

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-243. GPIO_INT_DISABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_INT_DISABLE* ioctl command disables an interrupt request generated by a GPIO pin by modifying the content of the Interrupt Enable Register.

Returns: None.

Range Issues: None.

Special Issues: Only one pin on the GPIO port may be allowed to generate the interrupt at the same time to obtain proper functionality.

Design/Implementation: The *GPIO_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-220. GPIO_INT_DISABLE

```
ioctl( GPIO_B, GPIO_INT_DISABLE, BIT_1);
```

This code disables the interrupt request generated by pin 1 on Port B.

5.8.3.8 GPIO_INT_ENABLE - enable interrupt request generated by GPIO

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-244. GPIO_INT_ENABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_INT_ENABLE* ioctl command enables an interrupt request generated by a GPIO pin by modifying the content of the Interrupt Enable Register.

Returns: None.

Range Issues: None.

Special Issues: Only one pin on the GPIO port may be allowed to generate the interrupt at the same time to obtain proper functionality.

Design/Implementation: The *GPIO_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-221. GPIO_INT_ENABLE

```
ioctl( GPIO_B, GPIO_INT_ENABLE, BIT_1);
```

This code enables an interrupt request generated by pin 1 on Port B.

5.8.3.9 GPIO_PULLUP_DISABLE - disable pull-up when the selected pins are configured as GPIO inputs

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_PULLUP_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-245. GPIO_PULLUP_DISABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_PULLUP_DISABLE* ioctl command disables pull-up on the selected pins by modifying the content of the Pull-Up Enable Register.

Returns: None.

Range Issues: None.

Special Issues: Pull-up may be enabled only if the selected GPIO pins are configured as inputs (See Section 5.8.3.5).

Design/Implementation: The *GPIO_PULLUP_DISABLE* ioctl command is implemented as a macro.

Example 5-222. GPIO_PULLUP_DISABLE

```
ioctl( GPIO_B, GPIO_PULLUP_DISABLE, BIT_0 | BIT_1 | BIT_2);
```

This code disables pull-up on the pins 0, 1, 2 on Port B.

5.8.3.10 GPIO_PULLUP_ENABLE - enable pull-up when the selected pins are configured as GPIO inputs

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_PULLUP_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-246. GPIO_PULLUP_ENABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_PULLUP_ENABLE* ioctl command enables pull-up on the selected pins by modifying the content of the Pull-Up Enable Register.

Returns: None.

Range Issues: None.

Special Issues: Pull-up may be enabled only if the selected GPIO pins are configured as inputs (See Section 5.8.3.5).

Design/Implementation: The *GPIO_PULLUP_ENABLE* ioctl command is implemented as a macro.

Example 5-223. GPIO_PULLUP_ENABLE

```
ioctl( GPIO_B, GPIO_PULLUP_ENABLE, BIT_0 | BIT_1 | BIT_2);
```

This code enables pull-up on the pins 0, 1, 2 on Port B.

5.8.3.11 GPIO_CLEAR_SW_INT_PENDING - disable interrupt request generated by software

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_CLEAR_SW_INT_PENDING,
           UWord16 param);
```

Arguments:

Table 5-247. GPIO_CLEAR_SW_INT_PENDING ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_CLEAR_SW_INT_PENDING* ioctl command disables a software generated interrupt request by modifying the content of the Interrupt Assert Register.

Returns: None.

Range Issues: None.

Special Issues: This functionality of the GPIO is dedicated only for software testing. This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *GPIO_CLEAR_SW_INT_PENDING* ioctl command is implemented as a macro.

Example 5-224. GPIO_CLEAR_SW_INT_PENDING

```
ioctl( GPIO_B, GPIO_CLEAR_SW_INT_PENDING, BIT_1);
```

This code disables an interrupt request generated by pin 1 on Port B.

5.8.3.12 GPIO_SW_INT_ASSERT - enable interrupt request generated by software

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SW_INT_ASSERT,
           UWord16 param);
```

Arguments:

Table 5-248. GPIO_SW_INT_ASSERT ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SW_INT_ASSERT* ioctl command enables a software generated interrupt request by modifying the content of the Interrupt Assert Register.

Returns: None.

Range Issues: None.

Special Issues: This functionality of the GPIO is dedicated only for software testing. This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *GPIO_SW_INT_ASSERT* ioctl command is implemented as a macro.

Example 5-225. GPIO_SW_INT_ASSERT

```
ioctl( GPIO_B, GPIO_SW_INT_ASSERT, BIT_1);
```

This code enables an interrupt request generated by pin 1 on Port B.

5.8.3.13 GPIO_INT_DETECTION_ACTIVE_HIGH - set the selected GPIO pins to active high

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_INT_DETECTION_ACTIVE_HIGH, UWord16 param);
```

Arguments:

Table 5-249. GPIO_INT_DETECTION_ACTIVE_HIGH ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_INT_DETECTION_ACTIVE_HIGH* ioctl command sets the selected GPIO pins to be active high by modifying the content of the Interrupt Polarity Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_INT_DETECTION_ACTIVE_HIGH* ioctl command is implemented as a macro.

Example 5-226. GPIO_INT_DETECTION_ACTIVE_HIGH

```
ioctl( GPIO_B, GPIO_INT_DETECTION_ACTIVE_HIGH, BIT_1);
```

This code sets pin 1 on Port B to be active high.

5.8.3.14 GPIO_INT_DETECTION_ACTIVE_LOW - set the selected GPIO pins to active low

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_INT_DETECTION_ACTIVE_LOW,
           UWord16 param);
```

Arguments:

Table 5-250. GPIO_INT_DETECTION_ACTIVE_LOW ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_INT_DETECTION_ACTIVE_LOW* ioctl command sets the selected GPIO pins to be active low by modifying the content of the Interrupt Polarity Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_INT_DETECTION_ACTIVE_LOW* ioctl command is implemented as a macro.

Example 5-227. GPIO_INT_DETECTION_ACTIVE_LOW

```
ioctl( GPIO_B, GPIO_INT_DETECTION_ACTIVE_LOW, BIT_1);
```

This code sets pin 1 on Port B to be active low.

5.8.3.15 GPIO_CLEAR_INT_PENDING - clear interrupt request flags

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_CLEAR_INT_PENDING, UWord16 param);
```

Arguments:

Table 5-251. GPIO_CLEAR_INT_PENDING ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_CLEAR_INT_PENDING* ioctl command clears the selected interrupt request flags generated by the GPIO by writing ones to the Interrupt Edge Sensitive Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO_CLEAR_INT_PENDING ioctl command should be used within the interrupt service routines to clear the interrupt request flags.

Design/Implementation: The *GPIO_CLEAR_INT_PENDING* ioctl command is implemented as a macro.

Example 5-228. GPIO_CLEAR_INT_PENDING

```
ioctl( GPIO_B, GPIO_CLEAR_INT_PENDING, BIT_5);
```

This code clears the interrupt request flags from the GPIO Port B, pin 5.

```
UWord16 intFlags;

intFlags = ioctl( GPIO_A, READ_INT_PENDING_REG, NULL );
...
ioctl( GPIO_B, GPIO_CLEAR_INT_PENDING, intFlags);
```

This code clears the appropriate interrupt request flags from the GPIO Port B.

5.8.3.16 GPIO_SET_PIN - set the selected GPIO pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SET_PIN, UWord16 param);
```

Arguments:

Table 5-252. GPIO_SET_PIN ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_PIN* ioctl command sets the selected GPIO pins by modifying the content of the Data Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6).

Design/Implementation: The *GPIO_SET_PIN* ioctl command is implemented as a macro.

Example 5-229. GPIO_SET_PIN

```
ioctl( GPIO_B, GPIO_SET_PIN, BIT_0 | BIT_1 | BIT_2);
```

This code sets pins 0, 1 and 2 on the GPIO, Port B.

5.8.3.17 GPIO_CLEAR_PIN - clear the selected GPIO pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_CLEAR_PIN, UWord16 param);
```

Arguments:

Table 5-253. GPIO_CLEAR_PIN ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_CLEAR_PIN* ioctl command clears the selected GPIO pins by modifying the content of the Data Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6).

Design/Implementation: The *GPIO_CLEAR_PIN* ioctl command is implemented as a macro.

Example 5-230. GPIO_CLEAR_PIN

```
ioctl( GPIO_B, GPIO_CLEAR_PIN, BIT_0 | BIT_1 | BIT_2);
```

This code clears pins 0, 1 and 2 on the GPIO, Port B.

5.8.3.18 GPIO_TOGGLE_PIN - toggle the selected GPIO pins

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_TOGGLE_PIN,
           UWord16 param);
```

Arguments:

Table 5-254. GPIO_TOGGLE_PIN ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_TOGGLE_PIN* ioctl command toggles the selected GPIO pins by modifying the content of the Data Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6).

Design/Implementation: The *GPIO_TOGGLE_PIN* ioctl command is implemented as a macro.

Example 5-231. GPIO_TOGGLE_PIN

```
ioctl( GPIO_B, GPIO_TOGGLE_PIN, BIT_0 | BIT_1 | BIT_2);
```

This code toggles pins 0, 1 and 2 on the GPIO, Port B.

5.8.3.19 GPIO_READ_DATA - read from the selected GPIO port

Call(s):

```
UWord16 ioctl(const int *pModuleBase, GPIO_READ_DATA, NULL);
```

Arguments:

Table 5-255. GPIO_READ_DATA ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.

Description: The *GPIO_READ_DATA* ioctl command returns the whole GPIO port by reading the GPIO Data Register.

Returns: None.

Special Issues: None.

Design/Implementation: The *GPIO_READ_DATA* ioctl command is implemented as a macro.

Example 5-232. GPIO_READ_DATA

```
tmp = ioctl( GPIO_B, GPIO_READ_DATA, NULL);
```

This code reads from GPIO Port B into a variable *tmp*.

5.8.3.20 GPIO_WRITE_DATA - write to the selected GPIO port

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_WRITE_DATA,
           UWord16 param);
```

Arguments:

Table 5-256. GPIO_WRITE_DATA ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	UWord16

Description: The *GPIO_WRITE_DATA* ioctl command writes the whole port by writing to the GPIO Data Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6).

Design/Implementation: The *GPIO_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-233. GPIO_WRITE_DATA

```
ioctl( GPIO_B, GPIO_WRITE_DATA, 0x0047);
```

This code writes the value *0x0047* into the GPIO Port B.

5.8.3.21 GPIO_READ_INT_PENDING_REG - read GPIO Interrupt Pending Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, GPIO_READ_INT_PENDING_REG,
          NULL);
```

Arguments:

Table 5-257. GPIO_READ_INT_PENDING_REG ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.

Description: The *GPIO_READ_INT_PENDING_REG* ioctl command reads the whole content of the Interrupt Pending Register.

Returns: content of the Interrupt Pending Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_READ_INT_PENDING_REG* ioctl command is implemented as a macro.

Example 5-234. GPIO_READ_INT_PENDING_REG

```
UWord16    tmp;

tmp = ioctl( GPIO_B, GPIO_READ_INT_PENDING_REG, NULL);
```

This code reads the GPIO Port B Interrupt Pending Register into a variable *tmp*.

5.8.3.22 GPIO_GET_INT_PENDING_FLAG - get the GPIO interrupt pending flag(s)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, GPIO_GET_INT_PENDING_FLAG,
          UWord16 param);
```

Arguments:

Table 5-258. GPIO_GET_INT_PENDING_FLAG ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_GET_INT_PENDING_FLAG* ioctl command reads the selected interrupt pending flag(s) from the Interrupt Pending Register.

Returns: the interrupt pending flag(s) status in its original bit position as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_GET_INT_PENDING_FLAG* ioctl command is implemented as a macro.

Example 5-235. GPIO_GET_INT_PENDING_FLAG

```
UWord16    tmp;

tmp = ioctl( GPIO_B, GPIO_GET_INT_PENDING_FLAG, BIT_0 | BIT_1);
```

This code reads the GPIO Port B pin/bit 0 and 1 interrupt pending flags into a variable *tmp*.

5.8.3.23 GPIO_TEST_INT_PENDING - test the GPIO interrupt pending flag(s)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, GPIO_TEST_INT_PENDING,
          UWord16 param);
```

Arguments:

Table 5-259. GPIO_TEST_INT_PENDING ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_TEST_INT_PENDING* ioctl command tests the selected interrupt pending flag(s) from the Interrupt Pending Register.

Returns: the non-zero value if the selected interrupt pending flag(s) is(are) set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_TEST_INT_PENDING* ioctl command is implemented as a macro.

Example 5-236. GPIO_TEST_INT_PENDING

```
UWord16    tmp;

tmp = ioctl( GPIO_B, GPIO_TEST_INT_PENDING, BIT_0 | BIT_1);
```

This code tests the GPIO Port B pin/bit 0 and 1 interrupt pending flags and writes the result into a variable *tmp*.

5.8.3.24 GPIO_SETAS_PUSHPULL - set the output driver of the selected pins to push-pull mode

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_PUSHPULL,
           UWord16 param);
```

Arguments:

Table 5-260. GPIO_SETAS_PUSHPULL ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_PUSHPULL* ioctl command sets the output driver of the selected GPIO pins to push-pull mode by modifying the content of the Push-Pull Mode Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *GPIO_SETAS_PUSHPULL* ioctl command is implemented as a macro.

Example 5-237. GPIO_SETAS_PUSHPULL

```
ioctl( GPIO_B, GPIO_SETAS_PUSHPULL, BIT_0 | BIT_1 | BIT_2);
```

This code sets the output driver of pins 0, 1 and 2 on Port B to push-pull mode.

5.8.3.25 GPIO_SETAS_OPENDRAIN - set the output driver of the selected pins to open drain mode

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SETAS_OPENDRAIN,
           UWord16 param);
```

Arguments:

Table 5-261. GPIO_SETAS_OPENDRAIN ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SETAS_OPENDRAIN* ioctl command sets the output driver of the selected GPIO pins to open drain mode by modifying the content of the Push-Pull Mode Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F83xx, MC56F801x and MC56F802x/3x.

Design/Implementation: The *GPIO_SETAS_OPENDRAIN* ioctl command is implemented as a macro.

Example 5-238. GPIO_SETAS_OPENDRAIN

```
ioctl( GPIO_B, GPIO_SETAS_OPENDRAIN, BIT_0 | BIT_1 | BIT_2);
```

This code sets the output driver of pins 0, 1 and 2 on Port B to open drain mode.

5.8.3.26 GPIO_READ_RAW_DATA - read the logic value of each GPIO pin on the selected GPIO port

Call(s):

```
UWord16 ioctl(const int *pModuleBase, GPIO_READ_RAW_DATA, NULL);
```

Arguments:

Table 5-262. GPIO_READ_RAW_DATA ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.

Description: The *GPIO_READ_RAW_DATA* ioctl command returns the logic value of each GPIO pin on the selected GPIO port - even when pins are not in GPIO mode. This command reads the GPIO Raw Data Register.

Returns: the content of the GPIO Raw Data Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *GPIO_READ_RAW_DATA* ioctl command is implemented as a macro.

Example 5-239. GPIO_READ_RAW_DATA

```
UWord16    tmp;

tmp = ioctl( GPIO_B, GPIO_READ_RAW_DATA, NULL);
```

This code reads logic state of all pins from GPIO Port B into a variable *tmp*.

5.8.3.27 GPIO_SET_HIGH_DRIVE_STRENGTH - set the high strength of the selected GPIO pin output driver

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SET_HIGH_DRIVE_STRENGTH,
           UWord16 param);
```

Arguments:

Table 5-263. GPIO_SET_HIGH_DRIVE_STRENGTH ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_HIGH_DRIVE_STRENGTH* ioctl command sets the high strength (8mA) of the selected GPIO pin output driver by modifying the content of the Drive Strength Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6). This command is applicable only on MC56F80xx.

Design/Implementation: The *GPIO_SET_HIGH_DRIVE_STRENGTH* ioctl command is implemented as a macro.

Example 5-240. GPIO_SET_HIGH_DRIVE_STRENGTH

```
ioctl( GPIO_A, GPIO_SET_HIGH_DRIVE_STRENGTH, BIT_0 | BIT_1 |
BIT_2);
```

This code sets high drive strength on pins 0, 1 and 2 on the GPIO, Port A.

5.8.3.28 GPIO_SET_LOW_DRIVE_STRENGTH - set the low strength of the selected GPIO pin output driver

Call(s):

```
void ioctl(const int *pModuleBase, GPIO_SET_LOW_DRIVE_STRENGTH,
           UWord16 param);
```

Arguments:

Table 5-264. GPIO_SET_LOW_DRIVE_STRENGTH ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_LOW_DRIVE_STRENGTH* ioctl command sets the low strength (4mA) of the selected GPIO pin output driver by modifying the content of the Drive Strength Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6). This command is applicable only on MC56F80xx.

Design/Implementation: The *GPIO_SET_LOW_DRIVE_STRENGTH* ioctl command is implemented as a macro.

Example 5-241. GPIO_SET_LOW_DRIVE_STRENGTH

```
ioctl( GPIO_A, GPIO_SET_LOW_DRIVE_STRENGTH, BIT_0 | BIT_2 );
```

This code sets low drive strength on pins 0 and 2 on the GPIO, Port A.

5.8.3.29 GPIO_SET_LOW_PASS_FILTER_ENABLE - set the low pass filter of the selected GPIO pin

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_SET_LOW_PASS_FILTER_ENABLE,UWord16 param);
```

Arguments:

Table 5-265. GPIO_SET_LOW_PASS_FILTER_ENABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_LOW_PASS_FILTER_ENABLE* ioctl command sets the low pass input filter of the selected GPIO pin by modifying the content of the Input Filter Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO input pins (See Section 5.8.3.3, Section 5.8.3.5). This command is applicable only on MC56F800x.

Design/Implementation: The *GPIO_SET_LOW_PASS_FILTER_ENABLE* ioctl command is implemented as a macro.

Example 5-242. GPIO_SET_LOW_PASS_FILTER_ENABLE

```
ioctl( GPIO_A, GPIO_SET_LOW_PASS_FILTER_ENABLE, BIT_1 | BIT_3 );
```

This code sets low pass input filter on pins 1 and 3 on the GPIO, Port A.

5.8.3.30 GPIO_SET_LOW_PASS_FILTER_DISABLE - disable the low pass filter of the selected GPIO pin

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_SET_LOW_PASS_FILTER_DISABLE,UWord16 param);
```

Arguments:

Table 5-266. GPIO_SET_LOW_PASS_FILTER_DISABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_LOW_PASS_FILTER_DISABLE* ioctl command disables the low pass input filter of the selected GPIO pin by modifying the content of the Input Filter Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO input pins (See Section 5.8.3.3, Section 5.8.3.5). This command is applicable only on MC56F800x.

Design/Implementation: The *GPIO_SET_LOW_PASS_FILTER_DISABLE* ioctl command is implemented as a macro.

Example 5-243. GPIO_SET_LOW_PASS_FILTER_DISABLE

```
ioctl( GPIO_A, GPIO_SET_LOW_PASS_FILTER_DISABLE, BIT_1 | BIT_3 );
```

This code disables low pass input filter on pins 1 and 3 on the GPIO, Port A.

5.8.3.31 GPIO_SET_SLEW_RATE_FILTER_ENABLE - set the slew rate of the selected GPIO pin

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_SET_SLEW_RATE_FILTER_ENABLE,UWord16 param);
```

Arguments:

Table 5-267. GPIO_SET_SLEW_RATE_FILTER_ENABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_SLEW_RATE_FILTER_ENABLE* ioctl command sets the slew rate mode of the selected GPIO pin output driver by modifying the content of the Slew Rate Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6). This command is applicable only on MC56F800x.

Design/Implementation: The *GPIO_SET_SLEW_RATE_FILTER_ENABLE* ioctl command is implemented as a macro.

Example 5-244. GPIO_SET_SLEW_RATE_FILTER_ENABLE

```
ioctl( GPIO_A, GPIO_SET_SLEW_RATE_FILTER_ENABLE, BIT_1 | BIT_3 );
```

This code sets slew rate mode on pins 1 and 3 on the GPIO, Port A.

5.8.3.32 GPIO_SET_SLEW_RATE_FILTER_DISABLE - disable the slew rate of the selected GPIO pin

Call(s):

```
void ioctl(const int *pModuleBase,
           GPIO_SET_SLEW_RATE_FILTER_DISABLE,UWord16 param);
```

Arguments:

Table 5-268. GPIO_SET_SLEW_RATE_FILTER_DISABLE ioctl call arguments

*pModuleBase	in	GPIO port identifier. Use GPIO_A, GPIO_B, GPIO_C, GPIO_D, GPIO_E or GPIO_F. Note that not all GPIO ports are available on all chips.
param	in	Use BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_6 | BIT_7 | BIT_8 | BIT_9 | BIT_10 | BIT_11 | BIT_12 | BIT_13 | BIT_14 | BIT_15 to select required GPIO pins. Note that not all GPIO pins are available on all chips.

Description: The *GPIO_SET_SLEW_RATE_FILTER_DISABLE* ioctl command disables the slew rate mode of the selected GPIO pin output driver by modifying the content of the Slew Rate Control Register.

Returns: None.

Range Issues: None.

Special Issues: The GPIO pins must be set as GPIO output pins (See Section 5.8.3.3, Section 5.8.3.6). This command is applicable only on MC56F800x.

Design/Implementation: The *GPIO_SET_SLEW_RATE_FILTER_DISABLE* ioctl command is implemented as a macro.

Example 5-245. GPIO_SET_SLEW_RATE_FILTER_DISABLE

```
ioctl( GPIO_A, GPIO_SET_SLEW_RATE_FILTER_DISABLE, BIT_1 | BIT_3 );
```

This code disables slew rate mode on pins 1 and 3 on the GPIO, Port A.

5.8.4 GPIO Driver Application

The GPIO driver application is designed for Motorola/Freescale EVM's and it is intended to illustrate the usage of this driver by a real example. It also verifies the functionality by blinking user LEDs on the EVM.

The GPIO driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8346EVM\gpio_demo and consists of the application *gpio_demo.mcp* and the source code for the application *main.c*.

See Table 5-269 for the LED configuration, specific to the individual EVM's.

Table 5-269. EVMs configuration

EVM	User LEDs on GPIO	Push Button for interrupt generation
MC56F8006DEMO	LED1 (PA0), LED2 (PA1), LED4 (PA4)	push S1 (PB2) or S2 (PB3)
MC56F8013DEMO	LED1 (PA0), LED2 (PA1), LED4 (PA3)	push S1 (PB2) or S2 (PB3)
MC56F8323EVM	LED0 (PC0), LED1 (PC1), LED3 (PC3)	short GPIO Port A pins 0-1
56F8300DEMO	D10 (PA0), D1(PA1)	short GPIO Port C pins 2-3
MC56F8346EVM	LED0 (PC0), LED1 (PC1), LED3 (PC3)	short GPIO Port E pins 5-6
MC56F8346CB	LED0 (PC0), LED1 (PC1), LED3 (PC2)	short GPIO Port E pins 5-6
MC56F8357EVM	LED0 (PC0), LED1 (PC1), LED3 (PC3)	short GPIO Port E pins 5-6
MC56F8367EVM	LED0 (PC0), LED1 (PC1), LED3 (PC3)	short GPIO Port E pins 5-6

Example 5-246. GPIO Driver Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*   File generated by Graphical Configuration Tool
*
************************************************************************.*/
```

```
#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 52.000 MHz
    VCO frequency: 208.000 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.389 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                   0x0082
#define OCCS_PLLDB_INIT                   0x2019

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enable , Wait enable
         Ext. bus driven when inactive: Disable
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output: CLKO select: sys_clk (OCCS) , CLKOUT mode: Tristated
   SEMI - CS0: Base address: 0x0, Blocksize: 128K , BYTE_EN: Both bytes enable , R/W:
Read / Write
                PS/DS select: PS only , Write Wait States: 3, Read Wait States: 3
   SEMI - CS1: Base address: 0x0, Blocksize: 128K , BYTE_EN: Lower byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
   SEMI - CS2: Base address: 0x0, Blocksize: 128K , BYTE_EN: Upper byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS3: Base address: 0x0, Blocksize: 32K , BYTE_EN: Disable , R/W: Disable
                PS/DS select: Disable , Write Wait States: 23, Read Wait States: 23
.*/
#define SEMI_CSBAR0_INIT                  0x0005
#define SEMI_CSBAR1_INIT                  0x0005
#define SEMI_CSBAR2_INIT                  0x0005
#define SEMI_CSOR0_INIT                   0x1FC3
#define SEMI_CSOR1_INIT                   0x1BA3
#define SEMI_CSOR2_INIT                   0x1DA3
#define SIM_GPS_INIT                      0x0000

/*.
    INTC Configuration
```

```
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0000
#define INT_VECTOR_ADDR_31              gpioISR
#define INT_PRIORITY_LEVEL_31           INTC_LEVEL0

/*.
    GPIO_E Configuration
-----------------------------------------
    Pin  0: Function: TXD0 , PullUp: Enable ,
    Pin  1: Function: RXD0 , PullUp: Enable ,
    Pin  2: Function: A6 , PullUp: Enable ,
    Pin  3: Function: A7 , PullUp: Enable ,
    Pin  4: Function: SCLK0 , PullUp: Enable ,
    Pin  5: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Enable ,
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Open drain ,
    Pin  7: Function: SS0 , PullUp: Enable ,
    Pin  8: Function: GPIO , Direction: Input , PullUp: Disable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin 10: Function: GPIO , Direction: Input , PullUp: Disable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin 11: Function: GPIO , Direction: Input , PullUp: Disable , Interrupt: Disable,
Int.Polarity: Active high ,
.*/
#define GPIO_E_DDR_INIT                 0x0040
#define GPIO_E_PER_INIT                 0x009F
#define GPIO_E_IENR_INIT                0x0020


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-247. GPIO Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application which shows how to generate interrupt
*              from GPIO, LEDs are used as well
*
*              GPIO Port C configuration:
*                 pins 0, 1 - LED0, LED1 configured as outputs
*                 pin 3     - LED3, configured as output (toggled in interrupt)
*
*              GPIO Port E configuration:
*                 pins 5, 6 - short-circuit these to invoke the interrupt
*                             (SPI0 pins 1-2 on MC56F8346 EVM)
*                 see appconfig.h - GPIO_E Configuration section
*
* TARGET: MC56F8346 device
*
*****************************************************************************/

#include "qs.h"

#include "occs.h"
#include "sys.h"
#include "intc.h"
#include "gpio.h"
#include "cop.h"
#include "sci.h"
#include "spi.h"
#include "adc.h"
#include "qtimer.h"
#include "decoder.h"
#include "mc.h"
#include "pwm.h"

/*****************************************************************************
             interrupt service routine - toggles LED3
*****************************************************************************/

void gpioISR(void)
{
#pragma interrupt

             /* clear interrupt flags */
             ioctl(GPIO_E, GPIO_CLEAR_INT_PENDING, BIT_5);

             /* toggle red LED3 */
             ioctl(GPIO_C, GPIO_TOGGLE_PIN, BIT_3);
}

/*****************************************************************************
             main
*****************************************************************************/

void main(void)
{
             UWord32 i;
```

```
            /* configure GPIO C for LEDs */
    /* pins 0, 1, 3 as GPIO pins */
    ioctl(GPIO_C, GPIO_SETAS_GPIO, BIT_0 | BIT_1 | BIT_3 );

    /* pins 0, 1, 3 as output */
    ioctl(GPIO_C, GPIO_SETAS_OUTPUT, BIT_0 | BIT_1 | BIT_3);

    ioctl(GPIO_E, GPIO_INIT, NULL);
    ioctl(INTC, INTC_INIT, NULL);

            /* enable interrupts in SR */
            archEnableInt();

    /* toggle yellow LED1 */
    ioctl(GPIO_C, GPIO_TOGGLE_PIN, BIT_1 );

    while(1)
    {
            for(i=0; i<500000; i++) asm(nop);

                  /* toggles LED0 and LED1 */
            ioctl(GPIO_C, GPIO_TOGGLE_PIN, BIT_0 | BIT_1);
    }

}
```

5.9 ADC Driver (MC56F83xx,MC56F801x,MC56F802x/3x)

This section describes the API for the MC56F83xx and MC56F80xx ADC on-chip module. The functionality of the ADC module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual** and **56F802x/3x Peripheral Reference Manual**.

5.9.1 Introduction

The MC56F83xx devices can have up to two ADC modules (ADC A and ADC B). The MC56F80xx devices contains only one ADC_A module. The ADC module has 2x4 analog inputs, configurable as single-ended or differential. It can be configured for simultaneous sampling of the two inputs, for sequential scanning and storing of up to 8 measurements and for various other options. The ADC module can be synchronized with a PWM and a QTimer module. The ADC module can be powered down manually or automatically.

This section describes the Analog-to-Digital Converter driver software, which provides the lowest level software layer, interfacing hardware with software.

5.9.2 Quick Reference

This section is intended to be a source of quick access information, while the details are discussed in Section 5.9.3.

Table 5-270. ADC Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F832x	MC56F83xx
ADC (ADC_BASE)	0xF080	0xF080	N/A	N/A
ADC A (ADCA_BASE)	N/A	0xF200	0xF200	0xF200
ADC B (ADCB_BASE)	N/A	N/A	N/A	0xF240

5.9.2.1 API Definition

The following header files are needed in order to use the ADC device driver:

Required Header File(s):

```
#include "qs.h"
#include "adc.h"
```

The following information may be found in the header file *adc.h*.

Public Data Structure(s):

```
/* array of 8 result samples */
typedef UWord16 adc_tBuff[8];
```

5.9.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static ADC module configuration by the driver initialization routine. ADC_*x* should be replaced by ADC_A for the ADC A module and ADC_B for the ADC B module or by the ADC on MC56F80xx. These symbols are intended for the project-specific configuration file named *appconfig.h*. See e.g. Example 5-314 for more details.

Table 5-271. ADC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
ADC_*x*_ADCR1_INIT	UWord16	Initial value of the ADC_*x* Control Register 1
ADC_*x*_ADCR2_INIT	UWord16	Initial value of the ADC_*x* Control Register 2
ADC_*x*_ADZCC_INIT	UWord16	Initial value of the ADC_*x* Zero Crossing Control Register
ADC_*x*_ADLST1_INIT	UWord16	Initial value of the ADC_*x* Channel List Register 1
ADC_*x*_ADLST2_INIT	UWord16	Initial value of the ADC_*x* Channel List Register 2
only on MC56F802x/3x: ADC_*x*_ADLST3_INIT ADC_*x*_ADLST4_INIT	UWord16	Initial value of the ADC_*x* Channel List Register 3 and 4
ADC_*x*_ADSDIS_INIT	UWord16	Initial value of the ADC_*x* Sample Disable Register
ADC_*x*_ADCPOWER_INIT	UWord16	Initial value of the ADC_*x* Power Control Register
ADC_*x*_ADLLMT0_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 0
ADC_*x*_ADLLMT1_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 1
ADC_*x*_ADLLMT2_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 2
ADC_*x*_ADLLMT3_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 3
ADC_*x*_ADLLMT4_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 4
ADC_*x*_ADLLMT5_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 5
ADC_*x*_ADLLMT6_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 6
ADC_*x*_ADLLMT7_INIT	UWord16	Initial value of the ADC_*x* Low Limit Register 7
ADC_*x*_ADHLMT0_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 0
ADC_*x*_ADHLMT1_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 1
ADC_*x*_ADHLMT2_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 2
ADC_*x*_ADHLMT3_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 3
ADC_*x*_ADHLMT4_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 4

Table 5-271. ADC Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
ADC_*x*_ADHLMT5_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 5
ADC_*x*_ADHLMT6_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 6
ADC_*x*_ADHLMT7_INIT	UWord16	Initial value of the ADC_*x* High Limit Register 7
ADC_*x*_ADOFS0_INIT	UWord16	Initial value of the ADC_*x* Offset Register 0
ADC_*x*_ADOFS1_INIT	UWord16	Initial value of the ADC_*x* Offset Register 1
ADC_*x*_ADOFS2_INIT	UWord16	Initial value of the ADC_*x* Offset Register 2
ADC_*x*_ADOFS3_INIT	UWord16	Initial value of the ADC_*x* Offset Register 3
ADC_*x*_ADOFS4_INIT	UWord16	Initial value of the ADC_*x* Offset Register 4
ADC_*x*_ADOFS5_INIT	UWord16	Initial value of the ADC_*x* Offset Register 5
ADC_*x*_ADOFS6_INIT	UWord16	Initial value of the ADC_*x* Offset Register 6
ADC_*x*_ADOFS7_INIT	UWord16	Initial value of the ADC_*x* Offset Register 7
ADC_*x*_CAL_INIT	UWord16	Initial value of the ADC_*x* Calibration Register

5.9.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
```

Description: The *ioctl* call "changes" ADC device modes or accesses the ADC register(s). The third ioctl parameter is either a value or a pointer, depending on the Cmd type.

Arguments:

Table 5-272. ADC Driver Arguments - ioctl

pModuleBase	in	ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx.
cmd	in	Commands found in *adc.h* which are used to modify the ADC module status and control registers. See Table 5-273.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  combination of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-273. ioctl commands

Cmd	param	Return	Description
ADC_INIT	NULL	None	Initializes selected ADC module by data from configuration file (appconfig.h).
ADC_START	NULL **or on MC56F80xx:** ADC_CONVERTER_0 / ADC_CONVERTER_1	None	Starts ADC conversion in software.
ADC_STOP	ADC_ON / ADC_OFF **or on MC56F80xx:** ADC_ON_CONVERTER_0/ ADC_ON_CONVERTER_1/ ADC_OFF_CONVERTER_0/ ADC_OFF_CONVERTER_1	None	Places ADC to normal operation (ADC_ON) or to stop mode (ADC_OFF).
ADC_SYNC	ADC_ON / ADC_OFF **or on MC56F80xx:** ADC_ON_CONVERTER_0/ ADC_ON_CONVERTER_1/ ADC_OFF_CONVERTER_0/ ADC_OFF_CONVERTER_1	None	Selects ADC conversion START source - SYNC input (ADC_ON) or ADC_START command (ADC_OFF).
only on MC56F80xx: ADC_SIMULT	ADC_ON / ADC_OFF	None	Selects simultaneous or independent parallel scan.
ADC_SET_DIVISOR	UWord16	None	Sets ADC clock divisor select which determines ADC conversion speed.
ADC_SET_CHANNEL_CONFIG	ADC_AN0_AN1_SE | ADC_AN2_AN3_SE | ADC_AN4_AN5_SE | ADC_AN6_AN7_SE | ADC_AN0_AN1_DIFF | ADC_AN2_AN3_DIFF | ADC_AN4_AN5_DIFF | ADC_AN6_AN7_DIFF **different notation on MC56F80xx:** *ADC_ANAx_ANAy_zzz* *ADC_ANBx_ANBy_zzz*	None	Configures ADC analog inputs as single ended or differential in couples. Use appropriate combination of the predefined constants.
ADC_SET_SCAN_MODE	ADC_SCAN_ONCE_SEQUENTIAL / ADC_SCAN_ONCE_SIMULTANEOUS / ADC_SCAN_LOOP_SEQUENTIAL / ADC_SCAN_LOOP_SIMULTANEOUS / ADC_SCAN_TRIG_SEQUENTIAL / ADC_SCAN_TRIG_SIMULTANEOUS	None	Configures conversion sequence mode. It determines how conversion sequence is carried out.

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_WRITE_CHANNEL_LIST1 ***(obsolete, do not use in new applications)***	ADC_AN*x*_S*y* | ADC_AN*x*_S*y* | ADC_AN*x*_S*y* | ADC_AN*x*_S*y* *x* ... physical input number (0-7) *y* ... sample number (0-3)	None	Configures mapping of the physical inputs AN0-AN7 to lower four samples (SAMPLE0-3).
ADC_WRITE_CHANNEL_LIST2 ***(obsolete, do not use in new applications)***	ADC_AN*x*_S*y* | ADC_AN*x*_S*y* | ADC_AN*x*_S*y* | ADC_AN*x*_S*y* *x* ... physical input number (0-7) *y*... sample number (4-7)	None	Configures mapping of the physical inputs AN0-AN7 to higher four samples (SAMPLE4-7).
ADC_SET_LIST_SAMPLE0 ADC_SET_LIST_SAMPLE1 ADC_SET_LIST_SAMPLE2 ADC_SET_LIST_SAMPLE3 ADC_SET_LIST_SAMPLE4 ADC_SET_LIST_SAMPLE5 ADC_SET_LIST_SAMPLE6 ADC_SET_LIST_SAMPLE7 **on MC56F802x/3x only:** ADC_SET_LIST_SAMPLE8 ADC_SET_LIST_SAMPLE9 ADC_SET_LIST_SAMPLE10 ADC_SET_LIST_SAMPLE11 ADC_SET_LIST_SAMPLE12 ADC_SET_LIST_SAMPLE13 ADC_SET_LIST_SAMPLE14 ADC_SET_LIST_SAMPLE15	ADC_CH0 / ADC_CH1 / ADC_CH2 / ADC_CH3 / ADC_CH4 / ADC_CH5 / ADC_CH6 / ADC_CH7 **on MC56F80xx** ADC_ANA0 / ADC_ANA1 / ADC_ANA2 / ADC_ANA3 / ADC_ANB0 / ADC_ANB1 / ADC_ANB2 / ADC_ANB3 / **on MC56F802x/3x only:** ADC_ANA4 / ADC_ANA5 / ADC_ANA6 / ADC_ANA7 / ADC_ANB4 / ADC_ANB5 / ADC_ANB6 / ADC_ANB7 /	None	Sets mapping of the physical ADC input to the sample number 0...7 (0...15 on MC56F802x/3x)
ADC_WRITE_SAMPLE_DISABLE	**MC56F83xx only:** UWord16 / number of samples in scan sequence (1-8, 0-all samples disabled) MC56F80xx **only:** any combination of parameters: ADC_SAMPLE0 | ADC_SAMPLE1 | ADC_SAMPLE2 | ADC_SAMPLE3 | ADC_SAMPLE4 | ADC_SAMPLE5 | ADC_SAMPLE6 | ADC_SAMPLE7 | ADC_ENABLE_ALL **additional bits on MC56F802x/3x:** ADC_SAMPLE_8 ... ADC_SAMPLE_15	None	Sets the number(s) of samples in scan sequence.
ADC_WRITE_ZERO_CROSS_CNTRL	ADC_Sx_ZC_DISABLE / ADC_Sx_ZC_POSITIVE_NEGATIVE / ADC_Sx_ZC_NEGATIVE_POSITIVE / ADC_Sx_ZC_ANY_CROSS *x* ... sample number (0-7) - must be unique for each constant	None	Configures zero crossing detection logic for each sample independently. Each sample can be set to 1 of 4 modes.

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_ZERO_CROSS_CH0 ADC_ZERO_CROSS_CH1 ADC_ZERO_CROSS_CH2 ADC_ZERO_CROSS_CH3 ADC_ZERO_CROSS_CH4 ADC_ZERO_CROSS_CH5 ADC_ZERO_CROSS_CH6 ADC_ZERO_CROSS_CH7	ADC_ZC_DISABLE / ADC_ZC_POS2NEG / ADC_ZC_NEG2POS / ADC_ZC_BOTH	None	Configures zero crossing detection logic for sample 0.. 7.
ADC_END_OF_SCAN_INT	ADC_ENABLE / ADC_DISABLE **or on MC56F80xx:** ADC_ENABLE_CONVERTER_0 / ADC_ENABLE_CONVERTER_1 / ADC_DISABLE_CONVERTER_0 / ADC_DISABLE_CONVERTER_1	None	Enables/disables ADC End of Scan interrupt, this command has no effect in loop mode.
ADC_ZERO_CROSS_INT	ADC_ENABLE / ADC_DISABLE	None	Enables/disables ADC Zero Crossing interrupt.
ADC_LOW_LIMIT_INT	ADC_ENABLE / ADC_DISABLE	None	Enables/disables ADC Low Limit interrupt.
ADC_HIGH_LIMIT_INT	ADC_ENABLE / ADC_DISABLE	None	Enables/disables ADC High Limit interrupt.
ADC_INT_ENABLE ADC_INT_DISABLE	ADC_END_OF_SCAN | ADC_ZERO_CROSS | ADC_LOW_LIMIT | ADC_HIGH_LIMIT **additional bits on MC56F80xx:** ADC_END_OF_SCAN_CONVERTER_0| ADC_END_OF_SCAN_CONVERTER_1	None	Enables or disables the selected ADC interrupts.
ADC_TEST_INT_ENABLED	ADC_END_OF_SCAN | ADC_ZERO_CROSS | ADC_LOW_LIMIT | ADC_HIGH_LIMIT **additional bits on MC56F80xx:** ADC_END_OF_SCAN_CONVERTER_0| ADC_END_OF_SCAN_CONVERTER_1	UWord16	Returns non-zero if any of interrupts specified in parameter are enabled.
ADC_READ_SAMPLE	UWord16 (sample number 0-7)	Word16	Reads one sample result at a time. The sample number is determined by param.

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_READ_ALL_SAMPLES	adc_tBuff (Pointer to results buffer)	None	Reads all 8 sample results at a time. Sample results are copied to user allocated 8 word buffer. **note for MC56F802x/3x:** only the first 8 sample results are read
ADC_READ_STATUS	NULL	UWord16	Reads the ADC Status Register as is.
ADC_READ_LIMIT_STATUS	NULL	UWord16	Reads the ADC Limit Status Register as is.
ADC_READ_ZERO_CROSS_ STATUS	NULL	UWord16	Reads the ADC Zero Crossing Status Register as is.
ADC_GET_STATUS_CIP	NULL **or on MC56F80xx:** ADC_CONVERTER_0/ ADC_CONVERTER_1	UWord16	Reads the Status Register CIP bit (“Conversion/Scan in Progress”).
ADC_GET_STATUS_EOSI	NULL **or on MC56F80xx:** ADC_CONVERTER_0/ ADC_CONVERTER_1	UWord16	Reads the Status Register EOSI bit (“End of Scan IRQ flag”).
ADC_GET_STATUS_ZCI	NULL	UWord16	Reads the Status Register ZCI bit (“Zero Cross IRQ flag”).
ADC_GET_STATUS_LLMTI	NULL	UWord16	Reads the Status Register LLMTI bit (“Low Limit IRQ flag”).
ADC_GET_STATUS_HLMTI	NULL	UWord16	Reads the Status Register HLMTI bit (“High Limit IRQ flag”).
ADC_GET_STATUS_RDY	UWord16 (sample number 0-7) **on** MC56F802x/3x**:** sample 0-15	UWord16	Reads the Status Register RDYx bit (“Ready Sample x flag”).
ADC_GET_LIMIT_STATUS_LLS	UWord16 (sample number 0-7)	UWord16	Reads the Limit Status Register LLSx bit (“Low Limit Sample x flag”).
ADC_GET_LIMIT_STATUS_HLS	UWord16 (sample number 0-7)	UWord16	Reads the Limit Status Register HLSx bit (“High Limit Sample x flag”).

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_GET_ZERO_CROSS_ STATUS_ZCS	UWord16 (sample number 0-7)	UWord16	Reads the Zero Crossing Status Register ZCSx bit (“Zero Crossing x flag”).
ADC_CLEAR_STATUS_EOSI	NULL **or on MC56F80xx:** ADC_CONVERTER_0/ ADC_CONVERTER_1	None	Clears the Status Register EOSI bit (“End of Scan IRQ flag”).
ADC_CLEAR_STATUS_LLMTI	NULL	None	Clears the Status Register LLMTI bit (“Low Limit IRQ flag”). This flag bit is cleared indirectly through clearing of all Low Limit Status bits in Limit Status Register.
ADC_CLEAR_STATUS_HLMTI	NULL	None	Clears the Status Register HLMTI bit (“High Limit IRQ flag”). This flag bit is cleared indirectly through clearing of all High Limit Status bits in Limit Status Register.
ADC_CLEAR_STATUS_ZCI	NULL	None	Clears the Status Register ZCI bit (“Zero Crossing IRQ flag”). This flag bit is cleared indirectly through clearing of all Zero Crossing Status bits in Zero Crossing Status Register.
ADC_CLEAR_LIMIT_STATUS_ LLS	UWord16 (sample number 0-7)	None	Clears the Limit Status Register LLSx bit (“Low Limit x bit”).
ADC_CLEAR_LIMIT_STATUS_ HLS	UWord16 (sample number 0-7)	None	Clears the Limit Status Register HLSx bit (“High Limit x bit”).
ADC_CLEAR_LIMIT_STATUS_ BITS	UWord16 as any combination of ADC_LLSx or ADC_HLSx bits. ADC_LLS_ALL or ADC_HLS_ALL may also be specified.	None	Clears selected bits in the Limit Status Register.
ADC_CLEAR_ZERO_CROSS_ STATUS_ZCS	UWord16 (sample number 0-7)	None	Clears the Zero Crossing Status Register ZCSx bit (“Zero Crossing x bit”).

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_WRITE_OFFSET0 ADC_WRITE_OFFSET1 ADC_WRITE_OFFSET2 ADC_WRITE_OFFSET3 ADC_WRITE_OFFSET4 ADC_WRITE_OFFSET5 ADC_WRITE_OFFSET6 ADC_WRITE_OFFSET7	UWord16 (offset value)	None	Writes the ADC Offset Register 0-7. It directly sets up an offset value for SAMPLE0-7.
ADC_WRITE_LOW_LIMIT0 ADC_WRITE_LOW_LIMIT1 ADC_WRITE_LOW_LIMIT2 ADC_WRITE_LOW_LIMIT3 ADC_WRITE_LOW_LIMIT4 ADC_WRITE_LOW_LIMIT5 ADC_WRITE_LOW_LIMIT6 ADC_WRITE_LOW_LIMIT7	UWord16 (low limit value)	None	Writes the ADC Low Limit Registers 0-7. It directly sets up Low Limit value for SAMPLE0-7.
ADC_WRITE_HIGH_LIMIT0 ADC_WRITE_HIGH_LIMIT1 ADC_WRITE_HIGH_LIMIT2 ADC_WRITE_HIGH_LIMIT3 ADC_WRITE_HIGH_LIMIT4 ADC_WRITE_HIGH_LIMIT5 ADC_WRITE_HIGH_LIMIT6 ADC_WRITE_HIGH_LIMIT7	UWord16 (high limit value)	None	Writes the ADC High Limit Registers 0-7. It directly sets up High Limit value for SAMPLE0-7.
ADC_READ_LOW_LIMIT	UWord16 (sample number)	UWord16	Reads the ADC Low Limit Register x value. The sample number which low limit value belongs to is determined by param.
ADC_READ_HIGH_LIMIT	UWord16 (sample number)	UWord16	Reads the ADC HighLimit Register x value. The number of the sample which high limit value belongs to is determined by param.
ADC_READ_OFFSET	UWord16 (sample number)	UWord16	Reads the ADC Offset Register x value. The number of the sample which offset value belongs to is determined by param.
ADC_POWER_DOWN	ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF	None	Forces to power down individual ADC converters and the voltage reference.
ADC_POWER_UP	ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF	None	Forces to power up individual ADC converters and the voltage reference.
ADC_POWER_SAVE_MODE	ADC_ON / ADC_OFF	None	Controls ADC power savings mode.

Table 5-273. ioctl commands (Continued)

Cmd	param	Return	Description
ADC_SET_POWER_UP_DELAY	UWord16	None	Sets ADC power up delay.
ADC_GET_POWER_STATUS	ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF	None	Gets ADC converters and the voltage reference status.
ADC_READ_POWER_ CONTROL_REG	NULL	UWord16	Reads the ADC Power Control Register content.
only on MC56F83xx: ADC_CALIB_ENABLE	ADC_CONVERTER_0 | ADC_CONVERTER_1	None	Enables/enters the ADC calibration mode.
only on MC56F83xx: ADC_CALIB_DISABLE	ADC_CONVERTER_0 | ADC_CONVERTER_1	None	Disables the ADC calibration mode (enables ADC normal operation).
only on MC56F83xx: ADC_SET_CONVERTER0_ CALIB_REF	ADC_VCAL_L / ADC_VCAL_H	None	Selects voltage reference to be used during ADC0 calibration.
only on MC56F83xx: ADC_SET_CONVERTER1_ CALIB_REF	ADC_VCAL_L / ADC_VCAL_H	None	Selects voltage reference to be used during ADC1 calibration.
only on MC56F80xx: ADC_AUTO_POWER_DOWN	ADC_ON / ADC_OFF	None	Switches on/off ADC Auto Power Down saving mode.
only on MC56F80xx: ADC_AUTO_STANDBY	ADC_ON / ADC_OFF	None	Enables / disables auto standby mode.
only on MC56F801x: ADC_SET_VREFL_SOURCE ADC_SET_VREFH_SOURCE	ADC_VREF_SRC_EXTERNAL / ADC_VREF_SRC_INTERNAL	None	Selects the source of the VREFLO and VREFH reference inputs.
only on MC56F802x/3x: ADC_SET_VREFH1_SOURCE ADC_SET_VREFL1_SOURCE ADC_SET_VREFH0_SOURCE ADC_SET_VREFL0_SOURCE	ADC_VREF_SRC_EXTERNAL / ADC_VREF_SRC_INTERNAL	None	Selects the source of the VREFLO and VREFH reference inputs for individual sub-converters.
only on MC56F802x/3x: ADC_SET_CALIB_SOURCE	*combination of ANA7 and ANB7 modes:* (ADC_ANA7_NORMAL / ADC_ANA7_FROM_DAC0 /) / (ADC_ANB7_NORMAL / ADC_ANB7_FROM_DAC1 /)	None	Selects internal source of ANA7 and ANB7 inputs.

5.9.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrates the ioctl commands usage.

5.9.3.1 ADC_INIT - initialize ADC module

Call(s):

```
void ioctl(const int *pModuleBase, ADC_INIT, NULL);
```

Arguments:

Table 5-274. ADC_INIT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_INIT* ioctl command is used to initialize the selected ADC module. Initialization consists of writing the initialization values to the following ADC registers: ADC Control Register 1 (ADCR1), ADC Control Register 2 (ADCR2), ADC Zero Crossing Control Register (ADZCC), ADC Channel List Registers (ADLSTx), ADC Sample Disable Register (ADSDIS), ADC Low and High Limit Registers (ADLLMT0–7) & (ADHLMT0–7), ADC Offset Registers (ADOFS0–7), ADC Power Control Register (ADCPOWER) and ADC Calibration Register (ADC_CAL). Initialization values (configuration items) for each register are taken from *appconfig.h*, where each configuration item corresponds to one ADC register. If no initialization values is defined in *appconfig.h,* then no initialization is performed. It is not necessary to define the initialization values for all ADC registers in *appconfig.h*, define only those which are needed. For reference on symbols of configuration items see Section 5.9.2.2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_INIT* ioctl command is implemented as a function call.

Example 5-248. ADC_INIT

```
ioctl(ADC_A, ADC_INIT, NULL);
```

This code initializes the ADC module by values defined in appconfig.h. The appconfig.h file can be edited manually or modified by the Graphical Configuration Tool.

5.9.3.2 ADC_START - start scan conversion cycle

Call(s):

```
on MC56F83xx:
void ioctl(const int *pModuleBase, ADC_START, NULL);

on MC56F80xx:
void ioctl(const int *pModuleBase, ADC_START, UWord16 param);
```

Arguments:

Table 5-275. ADC_START ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on 56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select started converter. Use NULL on MC56F83xx. Use ADC_CONVERTER_0 to start converter 0 or ADC_CONVERTER_1 to start converter 1 on MC56F80xx in non-simultaneous mode.

Description: The *ADC_START* ioctl command starts the ADC conversion in software. It writes "1" to bit START0 or START1 of the ADC Control Register 1 or 2, depending on parameter value ADC_CONVERTER_0 or ADC_CONVERTER_1. When parameter is NULL, START0 bit is set in the ADC Control Register 1 which is a standard ADC start procedure for MC56F83xx ADC or MC56F80xx ADC operating in simultaneous mode.

Before starting the conversion, the ADC should be configured either by a static configuration or by ioctl commands. Also, if the state of the ADC is not known, the *ADC_STOP* command should be issued before any ADC configuration change.

Returns: None.

Range Issues: None

Special Issues: The ADC_CONVERTER_0 and ADC_CONVERTER_1 parameter values are applicable only on the MC56F80xx devices operating in non-simultaneous mode.

Design/Implementation: The *ADC_START* command is implemented as a macro.

Example 5-249. ADC_START

```
ioctl(ADC_A, ADC_START, NULL);
```

This code starts an ADC scan conversion cycle. Scan conversion cycles cannot be started if a previous scan is carried out. In such a case it is necessary to wait for the end of a current scan cycle or to issue the *ioctl(ADC_x, ADC_STOP, ADC_ON)* command followed by the *ioctl(ADC_x, ADC_STOP, ADC_OFF)* command, which suspends the current scan cycle.

5.9.3.3 ADC_STOP - suspend scan conversion cycle

Call(s):

```
void ioctl(const int *pModuleBase, ADC_STOP, UWord16 param);
```

Arguments:

Table 5-276. ADC_STOP ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ON to stop ADC operations or ADC_OFF to bring ADC to operation mode on MC56F83xx. On MC56F80xx when ADC operates in non-simultaneous mode, use ADC_ON_CONVERTER_0 / ADC_OFF_CONVERTER_0 parameter value to stop or run the sub-converter 0 and ADC_ON_CONVERTER_1 / ADC_OFF_CONVERTER_1 parameter value to stop or run the sub-converter 1.

Description: The *ADC_STOP* ioctl command suspends an ADC conversion scan cycle (param=ADC_ON) or brings it to the idle operational state (param=ADC_OFF).

Parameter values ADC_ON, ADC_OFF, ADC_ON_CONVERTER_0 and ADC_OFF_CONVERTER_0 affect the STOP0 bit in the ADC Control Register 1. Parameter values ADC_ON_CONVERTER_1 and ADC_OFF_CONVERTER_1 affect the STOP1 bit in the ADC Control Register 2.

Returns: None.

Range Issues: None

Special Issues: The ADC_ON_CONVERTER_1 and ADC_OFF_CONVERTER_1 parameters are applicable only on the MC56F80xx devices operating in non-simultaneous mode.

Design/Implementation: The *ADC_STOP* command is implemented as a macro.

Example 5-250. ADC_STOP

```
/* stop ADC operations */
ioctl(ADC_A, ADC_STOP, ADC_ON);
/* change needed ADC parameters */
/* ... */
/* re-enable ADC operation */
ioctl(ADC_A, ADC_STOP, ADC_OFF);
```

This code suspends an ADC scan conversion cycle if carried out. Then it enables an ADC operation again. It is useful to cancel an ADC operation when, for example, loop scan modes are selected or, when an ADC re-configuration is needed and the state of the ADC is not known.

5.9.3.4 ADC_SYNC - control ADC trigger source

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SYNC, UWord16 param);
```

Arguments:

Table 5-277. ADC_SYNC ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired ADC trigger source. Use ADC_ON when ADC is to be started by a SYNC input signal (typically generated from a QTimer module) or ADC_OFF when ADC is to be started manually by ADC_START command. On MC56F80xx when ADC operates in non-simultaneous mode, use ADC_ON_CONVERTER_0, ADC_OFF_CONVERTER_0, ADC_ON_CONVERTER_1 and ADC_OFF_CONVERTER_1 bit parameters to select the desired sub-converter trigger source.

Description: The *ADC_SYNC* ioctl command configures the source of an ADC conversion trigger as a SYNC input signal (when param = ADC_ON) or as a software-initiated start (when param = ADC_OFF). See the ADC_START ioctl command for more information about the software-initiated start of conversion.

Parameter values ADC_ON, ADC_OFF, ADC_ON_CONVERTER_0 and ADC_OFF_CONVERTER_0 affect the SYNC0 bit in the ADC Control Register 1. Parameter values ADC_ON_CONVERTER_1 and ADC_OFF_CONVERTER_1 affect the SYNC1 bit in the ADC Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: The ADC_ON_CONVERTER_1 and ADC_OFF_CONVERTER_1 parameters are applicable only on the MC56F80xx devices operating in non-simultaneous mode.

Design/Implementation: The *ADC_SYNC* command is implemented as a macro.

Example 5-251. ADC_SYNC

```
/* setup of the QTimer module to output SYNC impulses at desired
time intervals */
/* ... */

/* sets ADC scan conversion start by SYNC input */
ioctl(ADC_A, ADC_SYNC, ADC_ON);
/* ... */
/* test SAMPLE ready flags to read conversion results */
/* ... */
```

This code sets up the ADC for hardware conversion start.

5.9.3.5 ADC_SIMULT - select simultaneous or independent scan mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SIMULT, UWord16 param);
```

Arguments:

Table 5-278. ADC_SIMULT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx.
param	in	Use ADC_ON to set simultaneous mode or ADC_OFF to enable an independent mode.

Description: The *ADC_SIMULT* ioctl command sets/resets the ADC Control Register 2 bit SIMULT. When independent parallel scan mode is set (param ADC_OFF), converter 0 and converter 1 are controlled independently by their set of control bits in the ADC Control Register 1 and the ADC Control Register 2 respectively. The ADC is set to simultaneous parallel scan mode (default setting) when using the ADC_ON parameter, then the converter 0 and converter 1 are controlled by the control bits in the ADC Control Register 1.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on the MC56F80xx devices.

Design/Implementation: The *ADC_SIMULT* command is implemented as a macro.

Example 5-252. ADC_SIMULT

```
/* sets ADC independent parallel scan mode */
ioctl(ADC, ADC_SIMULT, ADC_OFF);
```

This code resets the SIMULT bit in the ADC Control Register 2 to enable the independent parallel scan mode.

5.9.3.6 ADC_SET_DIVISOR - control ADC clock source divisor

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_DIVISOR,
           UWord16 param);
```

Arguments:

Table 5-279. ADC_SET_DIVISOR ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired ADC clock divisor value.

Description: The *ADC_SET_DIVISOR* ioctl command sets the ADC clock divisor, which determines the ADC clock and consecutively the conversion speed. It sets up the ADC Control Register 2 - lowest 5bits. It must be set within the specified range of an ADC clock, according to the following formula:

$$\text{DivisorValue} = \frac{\text{IPclock}}{2 \cdot \text{ADCclock}} - 1$$

IPclock ... peripheral bus clock

ADCclock ... ADC clock

Returns: None.

Range Issues: DivisorValue: 0-31 but limited by the specified ADC clock range and the IPclock.

Special Issues: None.

Design/Implementation: The *ADC_SET_DIVISOR* command is implemented as a macro.

Example 5-253. ADC_SET_DIVISOR

```
/* sets ADC clock 5MHz if IPclock=40MHz,
     DivisorValue = 40/(2*5)-1 = 3 */
ioctl(ADC_A, ADC_SET_DIVISOR, 3);
```

This code sets up the ADC clock to maximum 5MHz for the fastest ADC operation on MC56F83xx devices.

5.9.3.7 ADC_SET_CHANNEL_CONFIG - control ADC input channel configurations

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CHANNEL_CONFIG,
           UWord16 param);
```

Arguments:

Table 5-280. ADC_SET_CHANNEL_CONFIG ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired channel single-ended/differential inputs configuration. Use combination of the predefined constants: (ADC_AN0_AN1_SE / ADC_AN0_AN1_DIFF) | (ADC_AN2_AN3_SE / ADC_AN2_AN3_DIFF) | (ADC_AN4_AN5_SE / ADC_AN4_AN5_DIFF) | (ADC_AN6_AN7_SE / ADC_AN6_AN7_DIFF). A different values are defined for MC56F801x: (ADC_ANA0_ANA1_SE / ADC_ANA0_ANA1_DIFF) | (ADC_ANA2_ANA3_SE / ADC_ANA2_ANA3_DIFF) | (ADC_ANB0_ANB1_SE / ADC_ANB0_ANB1_DIFF) | (ADC_ANB2_ANB3_SE / ADC_ANB2_ANB3_DIFF) and additional are defined for MC56F802x/3x (ADC_ANA4_ANA5_SE / ADC_ANA4_ANA5_DIFF) | (ADC_ANA6_ANA7_SE / ADC_ANA6_ANA7_DIFF) | (ADC_ANB4_ANB5_SE / ADC_ANB4_ANB5_DIFF) | (ADC_ANB6_ANB7_SE / ADC_ANB6_ANB7_DIFF)

Description: The *ADC_SET_CHANNEL_CONFIG* ioctl command sets the mode of the analog inputs independently in input couples (AN0-AN1, AN2-AN3, AN4-AN5, AN6-AN7). Each input couple can be configured as differential (ADC_ANx_ANy_DIFF) or single ended (ADC_ANx_ANy_SE).
This command writes directly to bits [7:4] of the ADC Control Register 1. On the MC56F802x/3x devices, this command writes also to bits [9:6] of the ADC Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_SET_CHANNEL_CONFIG* command is implemented as a macro.

Example 5-254. ADC_SET_CHANNEL_CONFIG

```
/* sets ADC inputs: AN0-AN1:differential, AN2-AN3:single ended,
AN4-AN5:single ended, AN6-AN7 differential */
```

```
ioctl(ADC_A, ADC_SET_CHANNEL_CONFIG, ADC_AN0_AN1_DIFF | \
ADC_AN2_AN3_SE | ADC_AN4_AN5_SE | ADC_AN6_AN7_DIFF);
```

This code sets up the ADC input to the desired measurement configuration.

5.9.3.8 ADC_SET_SCAN_MODE - control ADC conversion scan sequence type

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_SCAN_MODE,
           UWord16 param);
```

Arguments:

Table 5-281. ADC_SET_SCAN_MODE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired scan type. Use one of the predefined constants: ADC_SCAN_ONCE_SEQUENTIAL / ADC_SCAN_ONCE_SIMULTANEOUS / ADC_SCAN_LOOP_SEQUENTIAL / ADC_SCAN_LOOP_SIMULTANEOUS / ADC_SCAN_TRIG_SEQUENTIAL / ADC_SCAN_TRIG_SIMULTANEOUS

Description: The *ADC_SET_SCAN_MODE* ioctl command sets 1 of 6 possible scan modes. For more info see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**. This command writes directly to bits [2:0] of the ADC Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_SET_SCAN_MODE* command is implemented as a macro.

Example 5-255. ADC_SET_SCAN_MODE

```
/* sets ADC scan to "Triggered Sequential Mode" */
ioctl(ADC_A, ADC_SET_SCAN_MODE, ADC_SCAN_TRIG_SEQUENTIAL);
```

This code sets up the ADC_A input to the desired scan mode.

5.9.3.9 ADC_SET_LIST_SAMPLE*x* - map physical input to SAMPLE*x*

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_LIST_SAMPLEx,
           UWord16 param);
```

Arguments:

Table 5-282. ADC_SET_LIST_SAMPLE*x* ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired sample *x* to physical channel mapping configuration. Use one of the predefined constants: ADC_CH0 / ADC_CH1 / ADC_CH2 / ADC_CH3 / ADC_CH4 / ADC_CH5 / ADC_CH6 / ADC_CH7 A different values are defined for MC56F801x: ADC_ANA0 / ADC_ANA1 / ADC_ANA2 / ADC_ANA3 / ADC_ANB0 / ADC_ANB1 / ADC_ANB2 / ADC_ANB3 and additional are defined for MC56F802x/3x ADC_ANA4 / ADC_ANA5 / ADC_ANA6 / ADC_ANA7 / ADC_ANB4 / ADC_ANB5 / ADC_ANB6 / ADC_ANB7

Description: The *ADC_SET_LIST_SAMPLEx* ioctl command defines the assignment of the physical inputs to SAMPLE*x*. It is possible to configure freely any of the analog inputs to the desired SAMPLEx, but a limitation exists for the simultaneous mode or when the differential inputs are used. For more info see the MC56F8300 Peripheral User Manual or the **56F8000 Peripheral Reference Manual**. This command writes directly to the corresponding ADC Channel List Register.

Note, that *x* represents the SAMPLE number (0-7) and it is part of the command name. On the MC56F802x/3x devices, the SAMPLE number is in the range 0-15.

Returns: None.

Range Issues: None.

Special Issues: Limitation exists for input to samples assignment for simultaneous mode and differential inputs.

Design/Implementation: The ADC_SET_LIST_SAMPLE*x* command is implemented as a macro.

Example 5-256. ADC_SET_LIST_SAMPLE*x*

```
/* maps ADC inputs to samples: */
/* AN6 -> S7, AN7 -> S6 */
ioctl(ADC_A, ADC_SET_LIST_SAMPLE7, ADC_CH6);
ioctl(ADC_A, ADC_SET_LIST_SAMPLE6, ADC_CH7);
```

This code sets up the ADC physical inputs to map to the desired SAMPLE channels.

5.9.3.10 ADC_WRITE_SAMPLE_DISABLE - define number of samples in scan conversion cycle

Call(s):

```
void ioctl(const int *pModuleBase, ADC_WRITE_SAMPLE_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-283. ADC_WRITE_SAMPLE_DISABLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired sample number (0-7) which is to be disabled. On MC56F80xx, use value 0-7 or 0-15 on MC56F802x/3x. You may also use any combination of ADC_SAMPLE0, ADC_SAMPLE1, .. , ADC_SAMPLE7 (...ADC_SAMPLE15) to specify a sample to be disabled. Use the ADC_ENABLE_ALL value to enale all samples.

Description: The *ADC_WRITE_SAMPLE_DISABLE* ioctl command sets the ADC Sample Disable Register to determine which samples are included in the scan cycle. Althought the approachis the same, a physical sample disable registers are slightly different on MC56F83xx, MC56F801x and MC56F802x/3x. Please read the Peripheral Reference Manual for more details about disabling samples on a particluar device.

Returns: None.

Range Issues: See Peripheral Reference Manual.

Special Issues: None.

Design/Implementation: The *ADC_WRITE_SAMPLE_DISABLE* command is implemented as a macro.

Example 5-257. ADC_WRITE_SAMPLE_DISABLE

```
/* setups ADC for scanning of the SAMPLE0-SAMPLE4 provided that
ADC is not in parallel mode */
ioctl(ADC_A, ADC_WRITE_SAMPLE_DISABLE, 5);

/* the same code using parameters */
ioctl(ADC_A, ADC_WRITE_SAMPLE_DISABLE, ADC_SAMPLE5);
```

This code sets up the scan sequence of the length of 5 samples.

5.9.3.11 ADC_WRITE_ZERO_CROSS_CNTRL - define configuration of the zero crossing detection

Call(s):

```
void ioctl(const int *pModuleBase, ADC_WRITE_ZERO_CROSS_CNTRL,
           UWord16 param);
```

Arguments:

Table 5-284. ADC_WRITE_ZERO_CROSS_CNTRL ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired configuration of the zero crossing detection for each sample independently. Use combination of the predefined constants: ADC_Sx_ZC_DISABLE | ADC_Sx_ZC_POSITIVE_NEGATIVE | ADC_Sx_ZC_NEGATIVE_POSITIVE | ADC_Sx_ZC_ANY_CROSS *x* ... sample number (0-7) - must be unique for each constant

Description: The *ADC_WRITE_ZERO_CROSS_CNTRL* ioctl command defines the type of zero crossing detection. It is possible to freely configure any sample to the desired type of zero crossing detection (4 modes). For more info see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**. This command writes directly to the ADC Zero Crossing Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The ADC_WRITE_ZERO_CROSS_CNTRL command is implemented as a macro.

Example 5-258. ADC_WRITE_ZERO_CROSS_CNTRL

```
/* setup ADC zero crossing detection for SAMPLE0-7: */
/* SAMPLE0 - positive to negative change */
/* SAMPLE1 - zero crossing disabled */
/* SAMPLE2 - negative to positive change */
/* SAMPLE3 - any sign change */
/* SAMPLE4 - negative to positive change */
/* SAMPLE5 - zero crossing disabled */
/* SAMPLE6 - any sign change */
/* SAMPLE7 - positive to negative change */

ioctl(ADC_A, ADC_WRITE_ZERO_CROSS_CNTRL,
      ADC_S0_ZC_POSITIVE_NEGATIVE | ADC_S1_ZC_DISABLE |
      ADC_S2_ZC_NEGATIVE_POSITIVE | ADC_S3_ZC_ANY_CROSS |
      ADC_S4_ZC_NEGATIVE_POSITIVE | ADC_S5_ZC_DISABLE |
      ADC_S6_ZC_ANY_CROSS | ADC_S7_ZC_POSITIVE_NEGATIVE);
```

This code sets up the ADC_A zero crossing logic to the desired configuration.

5.9.3.12 ADC_ZERO_CROSS_CH*x* - configure the zero crossing detection logic for SAMPLE*x*

Call(s):

```
void ioctl(const int *pModuleBase, ADC_ZERO_CROSS_CHx,
           UWord16 param);
```

Arguments:

Table 5-285. ADC_ZERO_CROSS_CH0 ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired configuration of the zero crossing detection for sample *x*. Use one of the predefined constants: ADC_ZC_DISABLE / ADC_ZC_POS2NEG / ADC_ZC_NEG2POS / ADC_ZC_BOTH.

Description: The *ADC_ZERO_CROSS_CHx* ioctl command defines the type of zero crossing detection logic for SAMPLE*x*. It is possible to freely configure any sample to the desired type of zero crossing detection (4 modes). For more info see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**. This command writes directly to the ADC Zero Crossing Control Register.

Note, that *x* represents the SAMPLE number (0-7) and it is part of the command name.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The ADC_ZERO_CROSS_CH*x* command is implemented as a macro.

Example 5-259. ADC_ZERO_CROSS_CH*x*

```
/* setup ADC zero crossing detection for SAMPLE5: */
/* SAMPLE5 - zero crossing disabled */

ioctl(ADC_A, ADC_ZERO_CROSS_CH5, ADC_ZC_DISABLE);
```

This code disables the ADC_A zero crossing logic for SAMPLE5.

5.9.3.13 ADC_END_OF_SCAN_INT - enable/disable End of Scan interrupt

Call(s):

```
void ioctl(const int *pModuleBase, ADC_END_OF_SCAN_INT,
           UWord16 param);
```

Arguments:

Table 5-286. ADC_END_OF_SCAN_INT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ENABLE to enable End of Scan interrupt or ADC_DISABLE to disable it. On MC56F80xx, the converter 0 and converter 1 may work in independent parallel scan mode and thus the End Of Scan interrupt can be enabled or disabled for both converters independently by ADC_ENABLE_CONVERTER_0, ADC_ENABLE_CONVERTER_1, ADC_DISABLE_CONVERTER_0, ADC_DISABLE_CONVERTER_1.

Description: The *ADC_END_OF_SCAN_INT* ioctl command is used to enable/disable an End of Scan interrupt. It sets the value of the ADC Control Register 1/2 EOSI0/EOSI1 bits.

Use the parameters ADC_ENABLE, ADC_DISABLE, ADC_ENABLE_CONVERTER_0 and ADC_DISABLE_CONVERTER_0 parameters to set or reset the ADC Control Register 1 EOSI0 bit. Use ADC_ENABLE_CONVERTER_1 and ADC_DISABLE_CONVERTER_1 parameters to set or reset the ADC Control Register 2 EOSI1 bit.

Returns: None.

Range Issues: None.

Special Issues: The ADC_ENABLE_CONVERTER_1 and ADC_DISABLE_CONVERTER_1 parameters are applicable only on the MC56F80xx devices.

Design/Implementation: The *ADC_END_OF_SCAN_INT* command is implemented as a macro.

Example 5-260. ADC_END_OF_SCAN_INT

```
/* enable ADC end of scan interrupt */
ioctl(ADC_A, ADC_END_OF_SCAN_INT, ADC_ENABLE);
/* ... */
/* disable ADC end of scan interrupt */
ioctl(ADC_A, ADC_END_OF_SCAN_INT, ADC_DISABLE);
```

This code switches on/off the ADC_A end of scan interrupt.

5.9.3.14 ADC_ZERO_CROSS_INT - enable/disable Zero Crossing interrupt

Call(s):

```
void ioctl(const int *pModuleBase, ADC_ZERO_CROSS_INT,
           UWord16 param);
```

Arguments:

Table 5-287. ADC_ZERO_CROSS_INT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ENABLE to enable Zero Crossing interrupt or ADC_DISABLE to disable it.

Description: The *ADC_ZERO_CROSS_INT* ioctl command is used to enable or disable a Zero Crossing interrupt. It sets the value of the ADC Control Register 1, bit ZCIE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_ZERO_CROSS_INT* command is implemented as a macro.

Example 5-261. ADC_ZERO_CROSS_INT

```
/* enable ADC zero crossing interrupt */
ioctl(ADC_A, ADC_ZERO_CROSS_INT, ADC_ENABLE);
/* ... */
/* disable ADC zero crossing interrupt */
/ioctl(ADC_A, ADC_ZERO_CROSS_INT, ADC_DISABLE);
```

This code switches on/off the ADC_A zero crossing interrupt.

5.9.3.15 ADC_LOW_LIMIT_INT - enable/disable Low Limit interrupt

Call(s):

```
void ioctl(const int *pModuleBase, ADC_LOW_LIMIT_INT,
           UWord16 param);
```

Arguments:

Table 5-288. ADC_LOW_LIMIT_INT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ENABLE to enable Low Limit interrupt or ADC_DISABLE to disable it.

Description: The *ADC_LOW_LIMIT_INT* ioctl command is used to enable or disable a Low Limit interrupt. It sets the value of the ADC Control Register 1, bit LLMTIE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_LOW_LIMIT_INT* command is implemented as a macro.

Example 5-262. ADC_LOW_LIMIT_INT

```
/* enable ADC low limit interrupt */
ioctl(ADC_A, ADC_LOW_LIMIT_INT, ADC_ENABLE);
/* ... */
/* disable ADC low limit interrupt */
/ioctl(ADC_A, ADC_LOW_LIMIT_INT, ADC_DISABLE);
```

This code switches on/off the ADC_A low limit interrupt.

5.9.3.16 ADC_HIGH_LIMIT_INT - enable/disable High Limit interrupt

Call(s):

```
void ioctl(const int *pModuleBase, ADC_HIGH_LIMIT_INT,
           UWord16 param);
```

Arguments:

Table 5-289. ADC_LOW_LIMIT_INT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ENABLE to enable Low Limit interrupt or ADC_DISABLE to disable it.

Description: The *ADC_HIGH_LIMIT_INT* ioctl command is used to enable or disable a High Limit interrupt. It sets the value of the ADC Control Register 1, bit HLMTIE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_HIGH_LIMIT_INT* command is implemented as a macro.

Example 5-263. ADC_HIGH_LIMIT_INT

```
/* enable ADC high limit interrupt */
ioctl(ADC_A, ADC_HIGH_LIMIT_INT, ADC_ENABLE);
/* ... */
/* disable ADC high limit interrupt */
/ioctl(ADC_A, ADC_HIGH_LIMIT_INT, ADC_DISABLE);
```

This code switches on/off the ADC_A high limit interrupt.

5.9.3.17 ADC_INT_ENABLE - enable ADC interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, ADC_INT_ENABLE, UWord16 param);
```

Arguments:

Table 5-290. ADC_INT_ENABLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired ADC interrupt(s). Use a combination of these predefined constants: ADC_END_OF_SCAN | ADC_ZERO_CROSS | ADC_LOW_LIMIT | ADC_HIGH_LIMIT Additional constants are available for the MC56F80xx ADC module operating in non-simultaneous mode: ADC_END_OF_SCAN_CONVERTER_0 | ADC_END_OF_SCAN_CONVERTER_1

Description: The *ADC_INT_ENABLE* ioctl command enables the ADC interrupt(s). This command sets the corresponding interrupt enable bits in the ADC Control Register 1 and 2 (ADCR1 and ADCR2).

On MC56F80xx, the ADC_END_OF_SCAN and ADC_END_OF_SCAN_CONVERTER_0 parameters control the EOSIE0 bit in the ADCR1 Control Register. When the ADC operates in non-simultaneous mode, the parameter value ADC_END_OF_SCAN_CONVERTER_1 may be used to control the EOSIE1 bit in the ADCR2 which controls the sub-converter 1 end-of-scan interrupt.

The other interrupt bits work the same way for all devices: the ADC_ZERO_CROSS controls the Zero Crossing Interrupt Enable (ZCIE). The ADC_LOW_LIMIT controls the Low Limit Interrupt Enable (LLMTIE). The ADC_HIGH_LIMIT controls the High Limit Interrupt Enable (HLMTIE).

Returns: None.

Range Issues: None.

Special Issues: The ADC_END_OF_SCAN_CONVERTER_1 is applicable only onthe MC56F80xx devices when operating in non-simultaneous mode.

Design/Implementation: The ADC_INT_ENABLE command is implemented as a macro.

Example 5-264. ADC_INT_ENABLE

```
/* enable ADC high limit interrupt */
ioctl(ADC_A, ADC_INT_ENABLE, ADC_HIGH_LIMIT);
```

This code enables the ADC A high limit interrupt.

```
/* enable ADC high limit interrupt and end of scan interrupt */
ioctl(ADC_B, ADC_INT_ENABLE, ADC_HIGH_LIMIT | ADC_END_OF_SCAN);
```

This code enables the ADC B high limit and end of scan interrupts.

5.9.3.18 ADC_INT_DISABLE - disable ADC interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, ADC_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-291. ADC_INT_DISABLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired ADC interrupt(s). Use a combination of these predefined constants: ADC_END_OF_SCAN | ADC_ZERO_CROSS | ADC_LOW_LIMIT | ADC_HIGH_LIMIT Additional constants are available for the MC56F80xx ADC module operating in non-simultaneous mode: ADC_END_OF_SCAN_CONVERTER_0 | ADC_END_OF_SCAN_CONVERTER_1

Description: The *ADC_INT_DISABLE* ioctl command disables the ADC interrupt(s). Please see the ADC_INT_ENABLE ioctl command for more detailed parameter explanation.

Returns: None.

Range Issues: None.

Special Issues: The ADC_END_OF_SCAN_CONVERTER_1 is applicable only on the MC56F80xx devices when operating in non-simultaneous mode.

Design/Implementation: The ADC_INT_DISABLE command is implemented as a macro.

Example 5-265. ADC_INT_DISABLE

```
/* disable ADC high limit interrupt */
ioctl(ADC_A, ADC_INT_DISABLE, ADC_HIGH_LIMIT);
```

This code disables the ADC A high limit interrupt.

```
/* disable ADC high limit interrupt and end of scan interrupt */
ioctl(ADC_B, ADC_INT_DISABLE, ADC_HIGH_LIMIT | ADC_END_OF_SCAN);
```

This code disables the ADC B high limit and end of scan interrupts.

5.9.3.19 ADC_TEST_INT_ENABLED - test what ADC interrupts are enabled

Call(s):

```
void ioctl(const int *pModuleBase, ADC_TEST_INT_ENABLED,
           UWord16 param);
```

Arguments:

Table 5-292. ADC_TEST_INT_ENABLED ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the ADC interrupt(s) to be tested. Use a combination of these predefined constants: ADC_END_OF_SCAN | ADC_ZERO_CROSS | ADC_LOW_LIMIT | ADC_HIGH_LIMIT Additional constants are available for the MC56F80xx ADC module operating in non-simultaneous mode: ADC_END_OF_SCAN_CONVERTER_0 | ADC_END_OF_SCAN_CONVERTER_1

Description: The *ADC_TEST_INT_ENABLED* ioctl command tests if any of the ADC interrupts specified in the parameter value are enabled. Please see the ADC_INT_ENABLE ioctl command for a detailed parameter explanation.

Returns: None.

Range Issues: None.

Special Issues: The ADC_END_OF_SCAN_CONVERTER_1 is applicable only on the MC56F80xx device when operating in non-simultaneous mode.

Design/Implementation: The ADC_TEST_INT_ENABLED command is implemented as a macro.

Example 5-266. ADC_TEST_INT_ENABLED

```
if(ioctl(ADC_A, ADC_TEST_INT_ENABLED, ADC_HIGH_LIMIT))
{
      ...
}
```

This code tests if the High-Limit interrupt is enabled.

5.9.3.20 ADC_READ_SAMPLE - read one sample result

Call(s):

```
Word16 ioctl(const int *pModuleBase, ADC_READ_SAMPLE,
             UWord16 param);
```

Arguments:

Table 5-293. ADC_READ_SAMPLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the SAMPLE number. Use number in the range 0-7 on all devices or in the range 0-15 on MC56F802x/3x.

Description: The *ADC_READ_SAMPLE* ioctl command reads a SAMPLEx result. It doesn't check if a new sample is ready, i.e. to find out that a new sample is converted, it is necessary to use other commands. Data interpretation of the result depends on the value of the ADC Offset Register x (written by command ADC_WRITE_OFFSETx). This command directly reads the ADC Result Register x.

Returns: Sample result.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_SAMPLE* command is implemented as a macro.

Example 5-267. ADC_READ_SAMPLE

```
Word16 R3;
/* read SAMPLE3 result */
R3 = ioctl(ADC_A, ADC_READ_SAMPLE, 3);
```

This code reads the ADC_A SAMPLE3 result.

5.9.3.21 ADC_READ_ALL_SAMPLES - read all 8 samples

Call(s):

```
void ioctl(const int *pModuleBase, ADC_READ_ALL_SAMPLES,
           adc_tBuff* pParam);
```

Arguments:

Table 5-294. ADC_READ_ALL_SAMPLES ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
pParam	inout	Pointer to an array of 8 words to store sample results. Array should be allocated by a caller.

Description: The *ADC_READ_ALL_SAMPLES* ioctl command reads eight samples and writes them to a buffer, pointed by pParam. It reads eight samples regardless of the length of the scan sequence. It doesn't check if new samples are ready, i.e. to find out that all samples are converted, it is necessary to use other commands. It directly reads the ADC Result Register 0-7.

Returns: None.

Range Issues: None.

Special Issues: This command always reads only the first eight samples, even on the ADC module of MC56F802x/3x, where up to 16 samples may be available.

Design/Implementation: The *ADC_READ_ALL_SAMPLES* command is implemented as a function call.

Example 5-268. ADC_READ_ALL_SAMPLES

```
adc_tBuff ResBuffer; /* buffer for sample results */

/* read all samples */
ioctl(ADC_A, ADC_READ_ALL_SAMPLES, ResBuffer);
/* all sample results are in ResBuffer array */
```

This code reads all eight samples from result registers and places them to result buffer ResBuffer (it is an array with a length of eight words).

5.9.3.22 ADC_READ_STATUS - read the whole ADC Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_STATUS, NULL);
```

Arguments:

Table 5-295. ADC_READ_STATUS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_READ_STATUS* ioctl command reads the whole ADC Status Register "as is". It is useful for testing more than one status bit at a time. For a description of the Status Register, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**. For testing the status bits, the use of some predefined bit masks is recommended:

ADC_ADSTAT_CIP - "Conversion in Progress" (on MC56F83xx)

ADC_ADSTAT_CIP0 - "Conversion in Progress converter 0" (on MC56F80xx)

ADC_ADSTAT_CIP1 - "Conversion in Progress converter 1" (on MC56F80xx)

ADC_ADSTAT_EOSI - "End of Scan interrupt flag" (on MC56F83xx)

ADC_ADSTAT_EOSI0 - "End of Scan interrupt flag converter 0" (on MC56F80xx)

ADC_ADSTAT_EOSI1 - "End of Scan interrupt flag converter 1" (on MC56F80xx)

ADC_ADSTAT_ZCI - "Zero Crossing interrupt flag"

ADC_ADSTAT_LLMTI - "Low Limit interrupt flag"

ADC_ADSTAT_HLMTI - "High Limit interrupt flag"

ADC_ADSTAT_RDY*x* - to test Sample Ready flag (RDYx) for sample *x* (0-7)

Returns: ADC Status Register value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_STATUS* command is implemented as a macro.

Example 5-269. ADC_READ_STATUS

```
UWord16 Status;
/* read status register */
Status = ioctl(ADC_A, ADC_READ_STATUS, NULL);
/* test bits LLMTI, HLMTI (low and high limit flags for all
samples) */
if (Status & (ADC_LLMTI | ADC_HLMTI))
{
/* at least one of LLMTI,HLMTI bits is set */
/* result is out of desired range */
/* ... */
}
```

This code reads the whole ADC Status Register and tests if any result went beyond low or high limits.

5.9.3.23 ADC_READ_LIMIT_STATUS - read the whole ADC Limit Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_LIMIT_STATUS,
          NULL);
```

Arguments:

Table 5-296. ADC_READ_LIMIT_STATUS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_READ_LIMIT_STATUS* ioctl command reads the whole ADC Limit Status Register "as is". It is useful for testing more than one status bit at a time. For a description of the Limit Status Register see the **MC56F8300 Peripheral User Manual** and the **56F8000 Peripheral Reference Manual**. For testing the status bits of the respective samples, the use of some predefined bit masks is recommended:

ADC_HLS*x* - "High Limit flag for sample *x*"

ADC_LLS*x* - "Low Limit flag for sample *x*"

Returns: ADC Limit Status Register value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_LIMIT_STATUS* command is implemented as a macro.

Example 5-270. ADC_READ_LIMIT_STATUS

```
UWord16 LimitStatus;
/* read limit status register */
LimitStatus = ioctl(ADC_A, ADC_READ_LIMIT_STATUS, NULL);
/* test bits LLS2,LLS4,LLS7 (low limit flags for samples 2,4 and
7) and HLS1 (high limit flag for sample 1 */
if (LimitStatus & (ADC_LLS2 | ADC_LLS4 | ADC_LLS7 | ADC_HLS1))
{
/* at least one flag is set */
/* result is out of desired range */
/* ... */
}
/* test bits HLS0,HLS3, HLS6,HLS7 (high limit flags for samples
0,3,6 and 7) */
if (LimitStatus & (ADC_HLS0 | ADC_HLS3 | ADC_HLS6 | ADC_HLS7))
{
/* at least one flag is set - result is out of desired range */
/* ... */}
```

This code reads the whole ADC Limit Status register and tests independently if the results went beyond low or high limits.

5.9.3.24 ADC_READ_ZERO_CROSS_STATUS - read the whole ADC Zero Crossing Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_ZERO_CROSS_STATUS,
          NULL);
```

Arguments:

Table 5-297. ADC_READ_ZERO_CROSS_STATUS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_READ_ZERO_CROSS_STATUS* ioctl command reads the whole ADC Zero Crossing Status Register "as is". It is useful for testing more than one status bit at a time. For a description of the Zero Crossing Status Register see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**. For testing the status bits of the respective samples, the use of predefined bit masks is recommended:

ADC_ADZCSTAT_ZCS*x*- "Zero-crossing Status bit for sample *x*"

Returns: ADC Zero Crossing Status Register value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_ZERO_CROSS_STATUS* command is implemented as a macro.

Example 5-271. ADC_READ_ZERO_CROSS_STATUS

```
/* read limit status register and test ZCS0,2,6 bits */
/* it test zero crossing for samples 0,2,6 */
if (ioctl(ADC_A, ADC_READ_ZERO_CROSS_STATUS, NULL) &
      (ADC_ADZCSTAT_ZCS0 | ADC_ADZCSTAT_ZCS2 | ADC_ADZCSTAT_ZCS6))
{
/* at least one flag is set */
/* zero crossing occurred for at least one of the samples */
/* ... */
}
```

This code reads the whole ADC Zero Crossing Status Register and tests if a zero crossing occurred for any of samples 0, 2, 6.

5.9.3.25 ADC_GET_STATUS_CIP - read the ADC Status Register, bit CIP

Call(s):

```
on MC56F83xx:
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_CIP, NULL);

on MC56F80xx:
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_CIP,
              UWord16 param);
```

Arguments:

Table 5-298. ADC_GET_STATUS_CIP ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Use NULL on MC56F83xx. Use ADC_CONVERTER_0 to read the CIP0 bit and ADC_CONVERTER_1 to read CIP1 bit or combination on MC56F80xx.

Description: The *ADC_GET_STATUS_CIP* ioctl command reads the ADC Status Register, CIP*x* ("Conversion in progress") bits. If the parameter is NULL or ADC_CONVERTER_0 this command reads the CIP (CIP0) bit in the ADC Status Register. If the parameter is ADC_CONVERTER_1 this command reads the CIP1 bit. For a more detailed description of the CIP/CIP0 and CIP1 bits and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if at least one of requested CIP0/CIP1 bits is set, zero otherwise.

Range Issues: None.

Special Issues: The ADC_CONVERTER_1 parameter is applicable only on MC56F80xx.

Design/Implementation: The *ADC_GET_STATUS_CIP* command is implemented as a macro.

Example 5-272. ADC_GET_STATUS_CIP

```
/* test status register CIP/CIP0 bit */
if (ioctl(ADC_A, ADC_GET_STATUS_CIP, NULL))
{
/* "scan conversion" is still in progress */
/* ... */
}
else
{
/* conversion finished */
/* ... */
}
```

This code tests the status register, bit CIP/CIP0 and shows the use of the command as a Boolean expression.

5.9.3.26 ADC_GET_STATUS_EOSI - read the ADC Status Register, bit EOSI

Call(s):

```
on MC56F83xx:
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_EOSI, NULL);

on MC56F80xx:
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_EOSI,
              UWord16 param);
```

Arguments:

Table 5-299. ADC_GET_STATUS_EOSI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Use NULL on MC56F83xx. Use ADC_CONVERTER_0 to read the EOSI0 bit and ADC_CONVERTER_1 to read the EOSI1 bit or their combination on MC56F80xx.

Description: The *ADC_GET_STATUS_EOSI* ioctl command reads the ADC Status Register, the EOSI*x* ("End of Scan Interrupt flag") bits. If the parameter is NULL / ADC_CONVERTER_0 this command reads the EOSI / EOSI0 bit. If the parameter is ADC_CONVERTER_1 this command reads the EOSI1 bit. For a more detailed description of the EOSI/EOSI0 and EOSI1 bits and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if EOSI / EOSI0 bit or EOSI1 bit is set, zero otherwise.

Range Issues: None.

Special Issues: ADC_CONVERTER_1 parameter is applicable only on MC56F80xx.

Design/Implementation: The *ADC_GET_STATUS_EOSI* command is implemented as a macro.

Example 5-273. ADC_GET_STATUS_EOSI

```
/* test status register EOSI/EOSI0 bit */
if (ioctl(ADC_A, ADC_GET_STATUS_EOSI, NULL))
{
/* scan cycle has been completed */
/* ... */
}
else
{
/* scan cycle has not been completed yet */
/* ... */
}
```

This code tests the ADC Status Register, bit EOSI/EOSI0 and shows the use of the command as a Boolean expression.

5.9.3.27 ADC_GET_STATUS_ZCI - read the ADC Status Register, bit ZCI

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_ZCI, NULL);
```

Arguments:

Table 5-300. ADC_GET_STATUS_ZCI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_GET_STATUS_ZCI* ioctl command reads the ADC Status Register, the ZCI ("Zero Crossing Interrupt flag") bit. For a more detailed description of the ZCI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if ZCI bit is set, zero otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_GET_STATUS_ZCI* command is implemented as a macro.

Example 5-274. ADC_GET_STATUS_ZCI

```
/* test status register ZCI bit */
if (ioctl(ADC_A, ADC_GET_STATUS_ZCI, NULL))
{
/* zero crossing occurred (for any sample) */
/* ... */
}
else
{
/* no zero crossing occurred */
/* ... */
}
```

This code tests the ADC Status Register, bit ZCI and shows the use of the command as a Boolean expression.

5.9.3.28 ADC_GET_STATUS_LLMTI - read the ADC Status Register, bit LLMTI

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_LLMTI,
          NULL);
```

Arguments:

Table 5-301. ADC_GET_STATUS_LLMTI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_GET_STATUS_LLMTI* ioctl command reads the ADC Status Register, the LLMTI ("Low Limit Interrupt flag") bit. For a more detailed description of the LLMTI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the LLMTI bit is set, zero otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_GET_STATUS_LLMTI* command is implemented as a macro.

Example 5-275. ADC_GET_STATUS_LLMTI

```
/* test status register LLMTI bit */
if (ioctl(ADC_A, ADC_GET_STATUS_LLMTI, NULL))
{
/* one of sample went below low limit */
/* ... */
}
else
{
/* all samples were higher than low limits */
/* ... */
}
```

This code tests the ADC Status Register, bit LLMTI and shows the use of the command as a Boolean expression.

5.9.3.29 ADC_GET_STATUS_HLMTI - read the ADC Status Register, bit HLMTI

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_HLMTI,
          NULL);
```

Arguments:

Table 5-302. ADC_GET_STATUS_HLMTI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_GET_STATUS_HLMTI* ioctl command reads the ADC Status Register, the HLMTI ("High Limit Interrupt flag") bit. For a more detailed description of the HLMTI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the HLMTI bit is set, zero otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_GET_STATUS_HLMTI* command is implemented as a macro.

Example 5-276. ADC_GET_STATUS_HLMTI

```
/* test status register HLMTI bit */
if (ioctl(ADC_A, ADC_GET_STATUS_HLMTI, NULL))
{
/* one of sample went above high limit */
/* ... */
}
else
{
/* all samples were lower than high limits */
/* ... */
}
```

This code tests the ADC Status Register, bit HLMTI and shows the use of the command as a Boolean expression.

5.9.3.30 ADC_GET_STATUS_RDY - read the ADC Status Register, bit RDYx

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_STATUS_RDY,
          UWord16 param);
```

Arguments:

Table 5-303. ADC_GET_STATUS_RDY ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number. Use the range 0-15 on the MC56F802x/3x devices.

Description: The *ADC_GET_STATUS_RDY* ioctl command reads the desired ADC Status Register, the RDY*x* ("Sample *x* Ready flag") bit. For a more detailed description of the RDYx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the RDY*x* bit is set, zero otherwise.

Range Issues: Sample number: 0-7 or 0-15 on MC56F802x/3x.

Special Issues: None.

Design/Implementation: The *ADC_GET_STATUS_RDY* command is implemented as a macro.

Example 5-277. ADC_GET_STATUS_RDY

```
Word16 S;
/* test status register RDY3 bit - ready for SAMPLE3 */
if (ioctl(ADC_A, ADC_GET_STATUS_RDY, 3))
{
/* SAMPLE3 is ready to read */
S = ioctl(ADC_A, ADC_READ_SAMPLE, 3);
}
```

This code tests the ADC Status Register, bit RDY3 and reads SAMPLE3 if ready (if conversion for this sample ended).

5.9.3.31 ADC_GET_LIMIT_STATUS_LLS - read the ADC Limit Status Register, bit LLSx

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_LIMIT_STATUS_LLS,
          UWord16 param);
```

Arguments:

Table 5-304. ADC_GET_LIMIT_STATUS_LLS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_GET_LIMIT_STATUS_LLS* ioctl command reads the desired ADC Limit Status Register, the LLS*x* ("Low Limit Sample *x* flag") bit. For a more detailed description of the LLSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the LLS*x* bit is set, zero otherwise.

Range Issues: Sample number: 0-7.

Special Issues: None.

Design/Implementation: The *ADC_GET_LIMIT_STATUS_LLS* command is implemented as a macro.

Example 5-278. ADC_GET_LIMIT_STATUS_LLS

```
Frac16 S;
/* test limit status register LLS0 bit - low limit check for
SAMPLE0 */
if (ioctl(ADC_A, ADC_GET_LIMIT_STATUS_LLS, 0))
{
/* SAMPLE0 result is below low limit */
}
else
{
/* SAMPLE0 result is above low limit */
}
```

This code tests the ADC Limit Status Register, bit LLS0 and shows the use of the command as a Boolean expression.

5.9.3.32 ADC_GET_LIMIT_STATUS_HLS - read the ADC Limit Status Register, bit HLSx

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_LIMIT_STATUS_HLS,
          UWord16 param);
```

Arguments:

Table 5-305. ADC_GET_LIMIT_STATUS_HLS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_GET_LIMIT_STATUS_HLS* ioctl command reads the desired ADC Limit Status Register, the HLS*x* ("High Limit Sample *x* flag") bit. For a more detailed description of the HLSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the HLS bit is set, zero otherwise.

Range Issues: Sample number: 0-7.

Special Issues: None.

Design/Implementation: The *ADC_GET_LIMIT_STATUS_HLS* command is implemented as a macro.

Example 5-279. ADC_GET_LIMIT_STATUS_HLS

```
/* test limit status register HLS6 bit - high limit check for
SAMPLE6 */
if (ioctl(ADC_A, ADC_GET_LIMIT_STATUS_HLS, 6))
{
/* SAMPLE6 result is above high limit */
}
else
{
/* SAMPLE6 result is below low limit */
}
```

This code tests the ADC Limit Status Register, bit HLS1 and shows the use of the command as a Boolean expression.

5.9.3.33 ADC_GET_ZERO_CROSS_STATUS_ZCS - read the ADC Zero Crossing Status Register, bit ZCSx

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
      ADC_GET_ZERO_CROSS_STATUS_ZCS,
      UWord16 param);
```

Arguments:

Table 5-306. ADC_GET_ZERO_CROSS_STATUS_ZCS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_GET_ZERO_CROSS_STATUS_ZCS* ioctl command reads the desired ADC Limit Status Register, the ZCSx ("Zero Crossing Sample x flag") bit. For a more detailed description of the ZCSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: Non-zero if the ZCSx bit is set, zero otherwise.

Range Issues: Sample number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_GET_ZERO_CROSS_STATUS_ZCS* command is implemented as a macro.

Example 5-280. ADC_GET_ZERO_CROSS_STATUS_ZCS

```
/* test zero crossing status register ZCS2 bit - zero crossing
check for SAMPLE2 */
if (ioctl(ADC_A, ADC_GET_ZERO_CROSS_STATUS_ZCS, 2))
{
/* zero crossing occurred for SAMPLE2 */
}
else
{
/* no zero crossing for SAMPLE2 */
}
```

This code tests the ADC Limit Status Register, bit ZCS2 and shows the use of the command as a Boolean expression.

5.9.3.34 ADC_CLEAR_STATUS_EOSI - clear the ADC Status Register, bit EOSI

Call(s):

```
on MC56F83xx:
void ioctl(const int *pModuleBase, ADC_CLEAR_STATUS_EOSI, NULL);

on MC56F80xx:
void ioctl(const int *pModuleBase, ADC_CLEAR_STATUS_EOSI,
           UWord16 param);
```

Arguments:

Table 5-307. ADC_CLEAR_STATUS_EOSI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Use NULL on MC56F83xx. Use ADC_CONVERTER_0 to clear the EOSI0 bit and ADC_CONVERTER_1 to clear EOSI1 bit or combination on MC56F80xx.

Description: The *ADC_CLEAR_STATUS_EOSI* ioctl command clears the ADC Status Register, the EOSI0/EOSI1 ("End of Scan flag") bit. In fact this bit is cleared by writing "1" to it. This command clears the EOSI/EOSI0 bit if parameter is NULL or ADC_CONVERTER_0 and the EOSI1 bit if parameter is ADC_CONVERTER_1. For a more detailed description of the EOSI0/EOSI1 bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: None.

Special Issues: The ADC_CONVERTER_1 parameter is applicable only on MC56F80xx.

Design/Implementation: The *ADC_CLEAR_STATUS_EOSI* command is implemented as a macro.

Example 5-281. ADC_CLEAR_STATUS_EOSI

```
/* clear status register EOSI/EOSI0 bit */
ioctl(ADC_A, ADC_CLEAR_STATUS_EOSI, NULL);
```

This code clears the ADC Status Register, bit EOSI. This bit is not cleared automatically when an interrupt service routine is entered.

5.9.3.35 ADC_CLEAR_STATUS_LLMTI - clear the ADC Status Register, bit LLMTI

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_STATUS_LLMTI, NULL);
```

Arguments:

Table 5-308. ADC_CLEAR_STATUS_LLMTI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_CLEAR_STATUS_LLMTI* ioctl command clears the ADC Status Register, the LLMTI ("Low Limit Interrupt flag") bit. This flag bit is cleared indirectly through clearing of all Low Limit Status bits in the Limit Status Register. For a more detailed description of the LLMTI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_STATUS_LLMTI* command is implemented as a macro.

Example 5-282. ADC_CLEAR_STATUS_LLMTI

```
/* clear status register LLMTI bit */
ioctl(ADC_A, ADC_CLEAR_STATUS_LLMTI, NULL);
```

This code clears the ADC Status Register, bit LLMTI. This bit is not cleared automatically when an interrupt service routine is entered.

5.9.3.36 ADC_CLEAR_STATUS_HLMTI - clear the ADC Status Register, bit HLMTI

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_STATUS_HLMTI, NULL);
```

Arguments:

Table 5-309. ADC_CLEAR_STATUS_HLMTI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_CLEAR_STATUS_HLMTI* ioctl command clears the ADC Status Register, the HLMTI ("High Limit Interrupt flag") bit. This flag bit is cleared indirectly through clearing of all High Limit Status bits in the Limit Status Register. For a more detailed description of the HLMTI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_STATUS_HLMTI* command is implemented as a macro.

Example 5-283. ADC_CLEAR_STATUS_HLMTI

```
/* clear status register HLMTI bit */
ioctl(ADC_A, ADC_CLEAR_STATUS_HLMTI, NULL);
```

This code clears the ADC Status Register, bit HLMTI. This bit is not cleared automatically when an interrupt service routine is entered.

5.9.3.37 ADC_CLEAR_STATUS_ZCI - clear the ADC Status Register, bit ZCI

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_STATUS_ZCI, NULL);
```

Arguments:

Table 5-310. ADC_CLEAR_STATUS_ZCI ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_CLEAR_STATUS_ZCI* ioctl command clears the ADC Status Register, the ZCI ("Zero Crossing Interrupt flag") bit. This flag bit is cleared indirectly through clearing of all Zero Crossing Status bits in the Zero Crossing Status Register. For a more detailed description of the ZCI bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_STATUS_ZCI* command is implemented as a macro.

Example 5-284. ADC_CLEAR_STATUS_ZCI

```
/* clear status register ZCI bit */
ioctl(ADC_A, ADC_CLEAR_STATUS_ZCI, NULL);
```

This code clears the ADC Status Register, bit ZCI. This bit is not cleared automatically when an interrupt service routine is entered.

5.9.3.38 ADC_CLEAR_LIMIT_STATUS_LLS - clear the ADC Limit Status Register, bit LLSx

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_LIMIT_STATUS_LLS,
           UWord16 param);
```

Arguments:

Table 5-311. ADC_CLEAR_LIMIT_STATUS_LLS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_CLEAR_LIMIT_STATUS_LLS* ioctl command clears the ADC Limit Status Register, the LLSx ("Low Limit Sample x flag") bit. This bit is cleared by writing "1" to it. For a more detailed description of the LLSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: Sample x number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_LIMIT_STATUS_LLS* command is implemented as a macro.

Example 5-285. ADC_CLEAR_LIMIT_STATUS_LLS

```
/* clear limit status register LLS4 bit (low limit flag for sample
4) */
ioctl(ADC_A, ADC_CLEAR_LIMIT_STATUS_LLS, 4);
```

This code clears the ADC Limit Status Register, bit LLS4.

5.9.3.39 ADC_CLEAR_LIMIT_STATUS_HLS - clear the ADC Limit Status Register, bit HLSx

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_LIMIT_STATUS_HLS,
           UWord16 param);
```

Arguments:

Table 5-312. ADC_CLEAR_LIMIT_STATUS_HLS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_CLEAR_LIMIT_STATUS_HLS* ioctl command clears the ADC Limit Status Register, the HLSx ("High Limit Sample x flag") bit. This bit is cleared by writing "1" to it. For a more detailed description of the HLSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: Sample x number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_LIMIT_STATUS_HLS* command is implemented as a macro.

Example 5-286. ADC_CLEAR_LIMIT_STATUS_HLS

```
/* clear limit status register HLS5 bit (high limit flag for
sample 5) */
ioctl(ADC_A, ADC_CLEAR_LIMIT_STATUS_LLS, 5);
```

This code clears the ADC Limit Status Register, bit HLS5.

5.9.3.40 ADC_CLEAR_LIMIT_STATUS_BITS - clear bits in the ADC Limit Status Register

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CLEAR_LIMIT_STATUS_BITS,
           UWord16 param);
```

Arguments:

Table 5-313. ADC_CLEAR_LIMIT_STATUS_BITS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	UWord16 as any combination of the ADC_LLSx or ADC_HLSx bits. ADC_LLS_ALL or ADC_HLS_ALL may also be specified.

Description: The *ADC_CLEAR_LIMIT_STATUS_BITS* ioctl command clears bits in the ADC Limit Status Register. Unlike the two commands ADC_CLEAR_LIMIT_STATUS_HLS and ADC_CLEAR_LIMIT_STATUS_LLS, this command clears any combination of the limit status bits in the register. The bits which are to be cleared are specified in the parameter.

This command writes the parameter value directly to the ADC Limit Status Register.

Returns: None.

Range Issues: Sample x number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_LIMIT_STATUS_BITS* command is implemented as a macro.

Example 5-287. ADC_CLEAR_LIMIT_STATUS_BITS

```
ioctl(ADC_A, ADC_CLEAR_LIMIT_STATUS_LLS, ADC_LLS_ALL);
```

This code clears the all low-limit status bits in the ADC Limit Status Register.

5.9.3.41 ADC_CLEAR_ZERO_CROSS_STATUS_ZCS - clear the ADC Zero Crossing Status Register, bit ZCSx

Call(s):

```
void ioctl(const int *pModuleBase,
           ADC_CLEAR_ZERO_CROSS_STATUS_ZCS,
           UWord16 param);
```

Arguments:

Table 5-314. ADC_CLEAR_ZERO_CROSS_STATUS_ZCS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Sample x number. Use 0-7 for sample number.

Description: The *ADC_CLEAR_ZERO_CROSS_STATUS_ZCS* ioctl command clears the ADC Zero Crossing Status Register, the ZCSx ("Zero Crossing Sample x flag") bit. This bit is cleared by writing "1" to it. For a more detailed description of the ZCSx bit and associated timing, see the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual**.

Returns: None.

Range Issues: Sample x number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_CLEAR_ZERO_CROSS_STATUS_ZCS* command is implemented as a macro.

Example 5-288. ADC_CLEAR_ZERO_CROSS_STATUS_ZCS

```
/* clear zero crossing status register ZCS1 bit (zero crossing
flag for sample 1) */
ioctl(ADC_A, ADC_CLEAR_ZERO_CROSS_STATUS_ZCS, 1);
```

This code clears the ADC Zero Crossing Status Register, bit ZCS1.

5.9.3.42 ADC_WRITE_OFFSET*x* - write the ADC Offset Register

Call(s):

```
void ioctl(const int *pModuleBase, ADC_WRITE_OFFSETx,
           UWord16 param);
```

Arguments:

Table 5-315. ADC_WRITE_OFFSET*x* ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to write desired ADC offset value for sample *x*.

Description: The *ADC_WRITE_OFFSETx* ioctl command writes an ADC offset value *x*, which is subtracted from the conversion result prior to writing to the ADC Result Register *x*. It writes directly to the ADC Offset Register *x* (*x* means sample number).

Note, that *x* represents the SAMPLE number (0-7) and it is part of the command name.

Returns: None.

Range Issues: Param: 0x0000, 0x0008 - 0x7FF8; sample number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_WRITE_OFFSETx* command is implemented as a macro.

Example 5-289. ADC_WRITE_OFFSET*x*

```
/* writes offset value for SAMPLE0 (0x3FF8 to get signed results)
*/
ioctl(ADC_A, ADC_WRITE_OFFSET0, 0x3FF8);

/* writes offset value for SAMPLE3 (0x0000 to get unsigned
results) */
ioctl(ADC_A, ADC_WRITE_OFFSET3, 0x0000);
```

This code writes an offset values for SAMPLE0, 3. These values are subtracted from the respective conversion result prior to writing to the result register. This way, the output coding can be controlled or it is possible to correct the system offset of the whole ADC chain.

5.9.3.43 ADC_WRITE_LOW_LIMIT*x* - write the ADC Low Limit Register

Call(s):

```
void ioctl(const int *pModuleBase, ADC_WRITE_LOW_LIMITx,
           UWord16 param);
```

Arguments:

Table 5-316. ADC_WRITE_LOW_LIMIT*x* ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to write desired ADC low limit value for sample *x*.

Description: The *ADC_WRITE_LOW_LIMITx* ioctl command writes an ADC low limit value *x*. This value is used for a comparison operation with the conversion result of sample *x*. When the conversion result is lower than the respective low limit value, the LLSx bit is set. This command writes directly to the ADC Low Limit Register *x* (*x* means sample number).

Note, that *x* represents the SAMPLE number (0-7) and it is part of the command name.

Returns: None.

Range Issues: Param: 0x0000, 0x0008 - 0x7FF8; sample number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_WRITE_LOW_LIMITx* command is implemented as a macro.

Example 5-290. ADC_WRITE_LOW_LIMIT*x*

```
/* writes low limit value for SAMPLE0 (0x0040) */
ioctl(ADC_A, ADC_WRITE_LOW_LIMIT0, 0x0040);

/* writes low limit value for SAMPLE6 (0x0100) */
ioctl(ADC_A, ADC_WRITE_LOW_LIMIT6, 0x0100);
```

This code writes a low limit values for SAMPLE0,6. These values are used to check the desired threshold. Values in result registers are not changed with this checking, only low limit flags are set (LLSx).

5.9.3.44 ADC_WRITE_HIGH_LIMIT*x* - write the ADC High Limit Register

Call(s):

```
void ioctl(const int *pModuleBase, ADC_WRITE_HIGH_LIMITx,
           UWord16 param);
```

Arguments:

Table 5-317. ADC_WRITE_LOW_LIMIT*x* ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to write desired ADC high limit value for sample *x*.

Description: The *ADC_WRITE_HIGH_LIMITx* ioctl command writes an ADC high limit value *x*. This value is used for a comparison operation with the conversion result of sample *x*. When the conversion result is higher than the respective high limit value, the HLSx bit is set. This command writes directly to the ADC High Limit Register *x* (*x* means sample number).

Note, that *x* represents the SAMPLE number (0-7) and it is part of the command name.

Returns: None.

Range Issues: Param: 0x0000, 0x0008 - 0x7FF8; sample number: 0-7

Special Issues: None.

Design/Implementation: The *ADC_WRITE_HIGH_LIMITx* command is implemented as a macro.

Example 5-291. ADC_WRITE_HIGH_LIMIT*x*

```
/* writes high limit value for SAMPLE1 (0x7F80) */
ioctl(ADC_A, ADC_WRITE_HIGH_LIMIT1, 0x0040);

/* writes high limit value for SAMPLE2 (0x7FE0) */
ioctl(ADC_A, ADC_WRITE_HIGH_LIMIT2, 0x0100);
```

This code writes a high limit values for SAMPLE1,2. These values are used to check the desired threshold. Values in result registers are not changed with this checking, only high limit flags are set (HLSx).

5.9.3.45 ADC_READ_LOW_LIMIT - read the ADC Low Limit Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_LOW_LIMIT,
          UWord16 param);
```

Arguments:

Table 5-318. ADC_READ_LOW_LIMIT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter x to select number of the SAMPLE which low limit value belongs to. Use 0-7.

Description: The *ADC_READ_LOW_LIMIT* ioctl command reads the ADC Low Limit Register for the sample given by the parameter value.

Returns: Low limit value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_LOW_LIMIT* command is implemented as a macro.

Example 5-292. ADC_READ_LOW_LIMIT

```
UWord16 R5;
/* read low limit for SAMPLE5 */
R5 = ioctl(ADC_A, ADC_READ_LOW_LIMIT, 5);
```

This code reads the ADC low limit for SAMPLE5. This is useful, for example, to update a register value relatively to its previous value, without any additional memory storage.

5.9.3.46 ADC_READ_HIGH_LIMIT - read the ADC High Limit Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_HIGH_LIMIT,
          UWord16 param);
```

Arguments:

Table 5-319. ADC_READ_HIGH_LIMIT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter x to select number of the SAMPLE which high limit value belongs to. Use 0-7.

Description: The *ADC_READ_HIGH_LIMIT* ioctl command reads the ADC High Limit Register for the sample given by the parameter value.

Returns: High limit value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_HIGH_LIMIT* command is implemented as a macro.

Example 5-293. ADC_READ_HIGH_LIMIT

```
UWord16 R2;
/* read low limit for SAMPLE2 */
R2 = ioctl(ADC_A, ADC_READ_HIGH_LIMIT, 2);
```

This code reads the ADC high limit for SAMPLE2. This is useful, for example, to update a register value relatively to its previous value, without any additional memory storage.

5.9.3.47 ADC_READ_OFFSET - read the ADC Offset Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_OFFSET,
              UWord16 param);
```

Arguments:

Table 5-320. ADC_READ_OFFSET ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter x to select number of the SAMPLE which offset value belongs to. Use 0-7.

Description: The *ADC_READ_OFFSET* ioctl command reads the ADC Offset Register for the sample given by the parameter value.

Returns: Offset value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_OFFSET* command is implemented as a macro.

Example 5-294. ADC_READ_OFFSET

```
UWord16 R1;
/* read low limit for SAMPLE1 */
R1 = ioctl(ADC_A, ADC_READ_OFFSET, 1);
```

This code reads the ADC offset for SAMPLE1. This is useful, for example, to update a register value relatively to its previous value, without any additional memory storage.

5.9.3.48 ADC_POWER_DOWN - force to power down ADC converters and the voltage reference

Call(s):

```
void ioctl(const int *pModuleBase, ADC_POWER_DOWN, UWord16 param);
```

Arguments:

Table 5-321. ADC_POWER_DOWN ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired action. Use combination of these pre-defined constants: ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF.

Description: The *ADC_POWER_DOWN* ioctl command forces to power down the selected ADC converters and the voltage reference. This command sets corresponding bits in the ADC Power Control Register (ADCPOWER). These bits are the Power Down Converter 0 (PD0) - Bit 0, when ADC_CONVERTER_0 is used as a parameter, the Power Down Converter 1 (PD1) - Bit 1, when ADC_CONVERTER_1 is used as a parameter and the Power Down Voltage Reference (PD2) - Bit 2, when ADC_VOLTAGE_REF is used as a parameter. See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: Note that this command affects the PSM bit in the ADC Power Control Register.

Design/Implementation: The ADC_POWER_DOWN command is implemented as a macro.

Example 5-295. ADC_POWER_DOWN

```
ioctl(ADC_A, ADC_POWER_DOWN, ADC_VOLTAGE_REF);
```

This code forces to power down the ADC A voltage reference.

```
ioctl(ADC_B, ADC_POWER_DOWN, ADC_CONVERTER_0 | ADC_CONVERTER_1);
```

This code forces to power down the ADC B converter 0 and converter 1.

5.9.3.49 ADC_POWER_UP - force to power up ADC converters and the voltage reference

Call(s):

```
void ioctl(const int *pModuleBase, ADC_POWER_UP, UWord16 param);
```

Arguments:

Table 5-322. ADC_POWER_UP ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired action. Use combination of these pre-defined constants: ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF.

Description: The *ADC_POWER_UP* ioctl command forces to power up the selected ADC converters and the voltage reference. This command clears corresponding bits in the ADC Power Control Register (ADCPOWER). These bits are the Power Down Converter 0 (PD0) - Bit 0, when ADC_CONVERTER_0 is used as a parameter, the Power Down Converter 1 (PD1) - Bit 1, when ADC_CONVERTER_1 is used as a parameter and the Power Down Voltage Reference (PD2) - Bit 2, when ADC_VOLTAGE_REF is used as a parameter. See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The ADC_POWER_UP command is implemented as a macro.

Example 5-296. ADC_POWER_UP

```
ioctl(ADC_A, ADC_POWER_UP, ADC_VOLTAGE_REF);
```

This code forces to power up the ADC A voltage reference.

```
ioctl(ADC_B, ADC_POWER_UP, ADC_CONVERTER_0 | ADC_CONVERTER_1);
```

This code forces to power up the ADC B converter 0 and converter 1.

5.9.3.50 ADC_POWER_SAVE_MODE - switch on/off ADC power savings mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_POWER_SAVE_MODE,
           UWord16 param);
```

Arguments:

Table 5-323. ADC_POWER_SAVE_MODE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select desired action. Use ADC_ON to activate the ADC power savings mode or ADC_OFF to deactivate it.

Description: The *ADC_POWER_SAVE_MODE* ioctl command is used to activate/deactivate the ADC power savings mode. It sets/clears the PSM bit (Bit 3) in the ADC Power Control Register (ADCPOWER). See the **MC56F8300 Peripheral User Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: Note that the power savings mode is useful only in Once/Trigger mode. This command is applicable only on MC56F83xx.

Design/Implementation: The ADC_POWER_SAVE_MODE command is implemented as a macro.

Example 5-297. ADC_POWER_SAVE_MODE

```
ioctl(ADC_A, ADC_POWER_SAVE_MODE, ADC_ON);
```

This code activates the power savings mode on the ADC A module.

```
ioctl(ADC_B, ADC_POWER_SAVE_MODE, ADC_OFF);
```

This code deactivates the power savings mode on the ADC B module.

5.9.3.51 ADC_SET_POWER_UP_DELAY - set ADC power up delay

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_POWER_UP_DELAY,
           UWord16 param);
```

Arguments:

Table 5-324. ADC_SET_POWER_UP_DELAY ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Value representing power up delay in ADC clocks.

Description: The *ADC_SET_POWER_UP_DELAY* ioctl command writes the value (*param*) to the ADC Power Control Register (ADCPOWER) bits PUDELAY. This value represents the number of ADC clocks required for an ADC converter to exit from power down mode or begin conversions after START/SYNC in power saving mode. See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: *param* must be within <0x000, 0x003f>.

Special Issues: None.

Design/Implementation: The *ADC_SET_POWER_UP_DELAY* ioctl command is implemented as a macro.

Example 5-298. ADC_SET_POWER_UP_DELAY

```
ioctl(ADC_A, ADC_SET_POWER_UP_DELAY, 0x000f);
```

This code sets power up delay to 15 ADC clocks.

5.9.3.52 ADC_GET_POWER_STATUS - get ADC converters and the voltage reference status

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_GET_POWER_STATUS,
          UWord16 param);
```

Arguments:

Table 5-325. ADC_GET_POWER_STATUS ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the desired power status bit. Use combination of these predefined constants: ADC_CONVERTER_0 | ADC_CONVERTER_1 | ADC_VOLTAGE_REF.

Description: The *ADC_GET_POWER_STATUS* ioctl command returns the status of the selected power status bits in the ADC Power Control Register. These bits are the ADC Converter 0 Power Status (PSTS0) - Bit 10, when ADC_CONVERTER_0 is used as a parameter, the ADC Converter 1 Power Status (PSTS1) - Bit 11, when ADC_CONVERTER_1 is used as a parameter and the Voltage Reference Power Status (PSTS2) - Bit 12, when ADC_VOLTAGE_REF is used as a parameter.

Returns: value representing the status bit(s) in its original bit position as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The ADC_GET_POWER_STATUS command is implemented as a macro.

Example 5-299. ADC_GET_POWER_STATUS

```
UWord16 pwrStatus;
pwrStatus = ioctl(ADC_A, ADC_GET_POWER_STATUS, ADC_VOLTAGE_REF);
```

This code stores the ADC A voltage reference power status to variable pwrStatus.

```
pwrStatus = ioctl(ADC_B, ADC_GET_POWER_STATUS, ADC_CONVERTER_0 |
          ADC_CONVERTER_1);
```

This code stores the ADC B converter 0 power status and converter 1 power status to variable pwrStatus.

5.9.3.53 ADC_READ_POWER_CONTROL_REG - return the content of the ADC Power Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, ADC_READ_POWER_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-326. ADC_READ_POWER_CONTROL_REG ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx or ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.

Description: The *ADC_READ_POWER_CONTROL_REG* ioctl command returns the content of the ADC Power Control Register (ADCPOWER).

Returns: content of the ADC Power Control Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The ADC_READ_POWER_CONTROL_REG command is implemented as a macro.

Example 5-300. ADC_READ_POWER_CONTROL_REG

```
UWord16 cntrl;
cntrl = ioctl(ADC_A, ADC_READ_POWER_CONTROL_REG, NULL);
```

This code stores the content of the ADC Power Control Register to variable cntrl.

5.9.3.54 ADC_CALIB_ENABLE - enable/enter the ADC calibration mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CALIB_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-327. ADC_CALIB_ENABLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the ADC converter. Use the following predefined constants or their combination: ADC_CONVERTER_0 | ADC_CONVERTER_1.

Description: The *ADC_CALIB_ENABLE* ioctl command is used to enter the calibration mode of the selected ADC converter(s) and the calibration voltage is routed to the ADC input. This command sets corresponding bits in the ADC Calibration Register (ADC_CAL). These bits are the Calibrate ADC0 (CAL0) - Bit 0, when ADC_CONVERTER_0 is used as a parameter and the Calibrate ADC1 (CAL1) - Bit 2, when ADC_CONVERTER_1 is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is available on on the MC56F83xx devices.

Design/Implementation: The ADC_CALIB_ENABLE command is implemented as a macro.

Example 5-301. ADC_CALIB_ENABLE

```
ioctl(ADC_A, ADC_CALIB_ENABLE, ADC_CONVERTER_0);
```

This code enables calibration of the ADC A converter 0.

```
ioctl(ADC_B, ADC_CALIB_ENABLE, ADC_CONVERTER_0 |
     ADC_CONVERTER_1);
```

Enter the calibration mode of the ADC B converter 0 and converter 1 using this code.

5.9.3.55 ADC_CALIB_DISABLE - disable the ADC calibration mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_CALIB_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-328. ADC_CALIB_DISABLE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the ADC converter. Use the following predefined constants or their combination: ADC_CONVERTER_0 | ADC_CONVERTER_1.

Description: The *ADC_CALIB_DISABLE* ioctl command is used to turn off the calibration mode of the selected ADC converter(s), i.e. ADC starts normal operation. This command clears corresponding bits in the ADC Calibration Register (ADC_CAL). These bits are the Calibrate ADC0 (CAL0) - Bit 0, when ADC_CONVERTER_0 is used as a parameter and the Calibrate ADC1 (CAL1) - Bit 2, when ADC_CONVERTER_1 is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is available on on the MC56F83xx devices.

Design/Implementation: The ADC_CALIB_DISABLE command is implemented as a macro.

Example 5-302. ADC_CALIB_DISABLE

```
ioctl(ADC_A, ADC_CALIB_DISABLE, ADC_CONVERTER_0 |
      ADC_CONVERTER_1);
```

This code disables calibration of the ADC A converter 0. and 1 - starts ADC normal operation.

5.9.3.56 ADC_SET_CONVERTER0_CALIB_REF - select calibration voltage for the ADC converter 0

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CONVERTER0_CALIB_REF,
           UWord16 param);
```

Arguments:

Table 5-329. ADC_SET_CONVERTER0_CALIB_REF ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the calibration/reference voltage. Use one of these predefined constants: ADC_VCAL_L / ADC_VCAL_H.

Description: The *ADC_SET_CONVERTER0_CALIB_REF* ioctl command is used to choose which reference will be used during ADC calibration. This command sets or clears the Calibration Reference Select ADC0 bit (CRS0 - Bit 1) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is available on on the MC56F83xx devices.

Design/Implementation: The ADC_SET_CONVERTER0_CALIB_REF command is implemented as a macro.

Example 5-303. ADC_SET_CONVERTER0_CALIB_REF

```
ioctl(ADC_A, ADC_SET_CONVERTER0_CALIB_REF, ADC_VCAL_H);
```

This code selects V_{CAL_H} (=0.75 V_{REFH}) as the calibration reference for the ADC A converter 0.

5.9.3.57 ADC_SET_CONVERTER1_CALIB_REF - select calibration voltage for the ADC converter 1

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CONVERTER1_CALIB_REF,
           UWord16 param);
```

Arguments:

Table 5-330. ADC_SET_CONVERTER1_CALIB_REF ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC_A and ADC_B on MC56F83xx. Note that ADC_B is not available on certain chips.
param	in	Parameter to select the calibration/reference voltage. Use one of these predefined constants: ADC_VCAL_L / ADC_VCAL_H.

Description: The *ADC_SET_CONVERTER1_CALIB_REF* ioctl command is used to choose which reference will be used during ADC calibration. This command sets or clears the Calibration Reference Select ADC1 bit (CRS1 - Bit 3) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is available on on the MC56F83xx devices.

Design/Implementation: The ADC_SET_CONVERTER1_CALIB_REF command is implemented as a macro.

Example 5-304. ADC_SET_CONVERTER1_CALIB_REF

```
ioctl(ADC_A, ADC_SET_CONVERTER1_CALIB_REF, ADC_VCAL_L);
```

This code selects V_{CAL_L} (=0.25 V_{REFH}) as the calibration reference for the ADC A converter 1.

5.9.3.58 ADC_AUTO_POWER_DOWN - switch on/off ADC Auto Power Down saving mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_AUTO_POWER_DOWN,
           UWord16 param);
```

Arguments:

Table 5-331. ADC_POWER_SAVE_MODE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx.
param	in	Parameter to select desired action. Use ADC_ON to activate the ADC auto power down mode or ADC_OFF to deactivate it.

Description: The *ADC_AUTO_POWER_DOWN* ioctl command is used to activate/deactivate the ADC Auto Power Down Mode. It sets/clears the APD bit (Bit 3) in the ADC Power Control Register (ADCPOWER). See the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: Note that the Auto Power Down mode is useful only in Once/Trigger mode. This command is applicable only on MC56F80xx (and it has the same function and it affects the same bit as the ADC_POWER_SAVE_MODE ioctl command).

Design/Implementation: The ADC_AUTO_POWER_DOWN command is implemented as a macro.

Example 5-305. ADC_AUTO_POWER_DOWN

```
ioctl(ADC, ADC_AUTO_POWER_DOWN, ADC_ON);
```

This code activates the Auto Power Down power saving mode on the ADC module.

5.9.3.59 ADC_AUTO_STANDBY - switch on/off ADC Standby Mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_AUTO_STANDBY,
           UWord16 param);
```

Arguments:

Table 5-332. ADC_POWER_SAVE_MODE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F80xx.
param	in	Parameter to select desired action. Use ADC_ON to activate the ADC standby mode or ADC_OFF to deactivate it.

Description: The *ADC_AUTO_STANDBY* ioctl command is used to activate/deactivate the ADC Auto Standby Mode. It sets/clears the ASB bit (Bit 15) in the ADC Power Control Register (ADCPOWER). See the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: Note that the Auto Standby Mode is useful only in Once/Trigger mode and with conversion clocks above 100kHz. Auto Standby mode is ignored if Auto Power Down Mode is selected. This command is applicable only on MC56F80xx.

Design/Implementation: The ADC_AUTO_STANDBY command is implemented as a macro.

Example 5-306. ADC_AUTO_STANDBY

```
ioctl(ADC, ADC_AUTO_STANDBY, ADC_ON);
```

This code activates the Auto Standby power saving mode on the ADC module.

5.9.3.60 ADC_SET_VREFL_SOURCE - select source of the VREFLO reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFL_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-333. ADC_SET_VREFL_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F801x.
param	in	Use parameter to select the source of the VREFLO reference input. Connect the VREFLO reference input to the ANB2 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VSSA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFL_SOURCE* ioctl command is used to choose source to be applied to the VREFLO reference input. This command sets or clears the SEL_VREFLO bit (Bit14) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x.

Design/Implementation: The ADC_SET_VREFL_SOURCE command is implemented as a macro.

Example 5-307. ADC_SET_VREFL_SOURCE

```
ioctl(ADC, ADC_SET_VREFL_SOURCE, ADC_VREF_SRC_EXTERNAL);
```

This code sets the ANB2 pin as input of the VREFLO reference input (pin must be configured as peripheral in the GPIO module).

5.9.3.61 ADC_SET_VREFH_SOURCE - select the source of the VREFH reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFH_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-334. ADC_SET_VREFH_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F801x.
param	in	Use parameter to select the source of the VREFH reference input. Connect the VREFH reference input to the ANA2 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VDDA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFH_SOURCE* ioctl command is used to choose source to be applied to the VREFH reference input. This command sets or clears the SEL_VREFH bit (Bit15) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx.

Design/Implementation: The ADC_SET_VREFH_SOURCE command is implemented as a macro.

Example 5-308. ADC_SET_VREFH_SOURCE

```
ioctl(ADC, ADC_SET_VREFH_SOURCE, ADC_VREF_SRC_INTERNAL);
```

This code interconnects the internal VDDA voltage to the VREFH reference input.

5.9.3.62 ADC_SET_VREFH0_SOURCE - select the source of the VREFH0 reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFH0_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-335. ADC_SET_VREFH0_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F802x/3x.
param	in	Use parameter to select the source of the VREFH reference input on the ADC sub-converter 0. Connect the VREFH reference input to the ANA2 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VDDA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFH0_SOURCE* ioctl command is used to choose source to be applied to the VREFH reference input of the sub-converter 0. This command sets or clears the SEL_VREFH0 bit (Bit13) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The ADC_SET_VREFH0_SOURCE command is implemented as a macro.

Example 5-309. ADC_SET_VREFH0_SOURCE

```
ioctl(ADC, ADC_SET_VREFH0_SOURCE, ADC_VREF_SRC_INTERNAL);
```

This code interconnects the internal VDDA voltage to the VREFH reference input of the ADC sub-converter 0.

5.9.3.63 ADC_SET_VREFL0_SOURCE - select the source of the VREFLO0 reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFL0_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-336. ADC_SET_VREFL0_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F802x/3x.
param	in	Use parameter to select the source of the VREFLO reference input on the ADC sub-converter 0. Connect the VREFLO reference input to the ANA3 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VSSA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFL0_SOURCE* ioctl command is used to choose source to be applied to the VREFLO reference input of the sub-converter 0. This command sets or clears the SEL_VREFLO0 bit (Bit12) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The ADC_SET_VREFL0_SOURCE command is implemented as a macro.

Example 5-310. ADC_SET_VREFL0_SOURCE

```
ioctl(ADC, ADC_SET_VREFL0_SOURCE, ADC_VREF_SRC_INTERNAL);
```

This code interconnects the internal VSSA voltage to the VREFLO reference input of the ADC sub-converter 0.

5.9.3.64 ADC_SET_VREFH1_SOURCE- select the source of the VREFH1 reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFH1_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-337. ADC_SET_VREFH1_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F802x/3x.
param	in	Use parameter to select the source of the VREFH reference input on the ADC sub-converter 1. Connect the VREFLH reference input to the ANB2 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VDDA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFH1_SOURCE* ioctl command is used to choose source to be applied to the VREFH reference input of the sub-converter 1. This command sets or clears the SEL_VREFH1 bit (Bit15) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The ADC_SET_VREFH1_SOURCE command is implemented as a macro.

Example 5-311. ADC_SET_VREFH1_SOURCE

```
ioctl(ADC, ADC_SET_VREFH1_SOURCE, ADC_VREF_SRC_INTERNAL);
```

This code interconnects the internal VDDA voltage to the VREFH reference input of the ADC sub-converter 1.

5.9.3.65 ADC_SET_VREFL1_SOURCE - select the source of the VREFLO0 reference input

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VREFL1_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-338. ADC_SET_VREFL1_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F802x/3x.
param	in	Use parameter to select the source of the VREFLO reference input on the ADC sub-converter 1. Connect the VREFLO reference input to the ANB3 pin (parameter ADC_VREF_SRC_EXTERNAL) or to the internal VSSA voltage (parameter ADC_VREF_SRC_INTERNAL).

Description: The *ADC_SET_VREFL1_SOURCE* ioctl command is used to choose source to be applied to the VREFLO reference input of the sub-converter 1. This command sets or clears the SEL_VREFLO1 bit (Bit14) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The ADC_SET_VREFL1_SOURCE command is implemented as a macro.

Example 5-312. ADC_SET_VREFL1_SOURCE

```
ioctl(ADC, ADC_SET_VREFL1_SOURCE, ADC_VREF_SRC_INTERNAL);
```

This code interconnects the internal VSSA voltage to the VREFLO reference input of the ADC sub-converter 1.

5.9.3.66 ADC_SET_CALIB_SOURCE - select source of ANA7 and ANB7 signals

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CALIB_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-339. ADC_SET_CALIB_SOURCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC on MC56F802x/3x.
param	in	Use combination of modes for ANA7 and ANB7 pins. For each pin, use exactly one of available modes. You can also omit the value for either pin completely. (ADC_ANA7_NORMAL / ADC_ANA7_FROM_DAC0 /) | (ADC_ANB7_NORMAL / ADC_ANB7_FROM_DAC1 /)

Description: The *ADC_SET_CALIB_SOURCE* ioctl command is used to select what signal will be routed to the ABA7 or ANB7 ADC inputs. By default, the signal coming from external pin is selected (NORMAL mode), even if the pin is not available. The other choice is to use the DAC output as a ADC input.

This command sets or clears the SEL_DAC0 and SEL_DAC1 bits (Bit0 and 1) in the ADC Calibration Register (ADC_CAL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The ADC_SET_CALIB_SOURCE command is implemented as a macro.

Example 5-313. ADC_SET_CALIB_SOURCE

```
ioctl(ADC, ADC_SET_CALIB_SOURCE, ADC_ANA7_NORMAL |
      ADC_ANB7_NORMAL);
```

This code sets-up a normal routing of both ANA7 and ANB7 input signals.

5.9.4 ADC Driver Application

The ADC driver application is designed for Freescale/Motorola EVM's. It is intended to illustrate the usage of this driver by a real example. It's purpose is also to verify the functionality by an analog-to-digital conversion of the voltage, connected to ADC inputs. Some examples show ADC operations in following ways:

- software ADC conversion start, stop, results reading with polling interface

- hardware ADC conversion start, interrupt driven interface

The ADC driver applications can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\adc_demo*. It consists of the application project *adc_demo.mcp* and the source code for the application *main.c*.

Example 5-314. ADC Driver Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Thu, 08/Feb/2007, 10:50:21
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
```

```
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
.*/
#define SIM_GPS_INIT                        0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                    0x0005
#define SEMI_CSBAR1_INIT                    0x0005
#define SEMI_CSBAR2_INIT                    0x0005
#define SEMI_CSOR0_INIT                     0x1FC3
#define SEMI_CSOR1_INIT                     0x1BA3
#define SEMI_CSOR2_INIT                     0x1DA3

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                     0x1FC1

/*.
    ADC_A Configuration
-----------------------------------------
    Clock frequency: 5 MHz
    Trigger source: Software start
    Channel Configuration: AN0-AN1: Single ended , AN2-AN3: Single ended
```

```
                          AN4-AN5: Single ended , AN6-AN7: Single ended
    Channel to Sample Mapping: SMP0: AN0 , SMP1: AN1 , SMP2: AN2 , SMP3: AN3
                               SMP4: AN4 , SMP5: AN5 , SMP6: AN6 , SMP7: AN7
    Scan Mode: Once sequential
    Zero crossing mode: SMP0: Disabled , SMP1: Disabled , SMP2: Disabled , SMP3:
Disabled
                        SMP4: Disabled , SMP5: Disabled , SMP6: Disabled , SMP7:
Disabled
    Power saving mode: Disable
    Power down converter 0 (AN0-AN3): No
    Power down converter 1 (AN4-AN8): No
    Power down voltage reference: No
    Power up delay: 13, Delay: 2.6 us
    Interrupts: Conversion complete: Disabled
                High limit error: Disabled
                Low limit error: Disabled
                Zero crossing: Disabled
.*/
#define ADC_A_ADCR1_INIT                0x0000
#define ADC_A_ADLST1_INIT               0x3210
#define ADC_A_ADLST2_INIT               0x7654


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-315. ADC Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Demo program for ADC driver
*
* TARGET: MC56F83xx devices
*
*****************************************************************************/

#include "qs.h"

#include "sys.h"
#include "cop.h"
#include "adc.h"

/* results buffer */
adc_tBuff adc_Buff;


/*************************************************************
 * Example with polling access and statically configured ADC
 *************************************************************/

void main (void)
{
    volatile UWord16 w1;
```

```
    /* initialize watchdog and system modules */
    ioctl(COP, COP_INIT, NULL);
    ioctl(SYS, SYS_INIT, NULL);

    /* configure ADC converter according to appconfig.h init values */
    ioctl(ADC_A, ADC_INIT, NULL);

    /* start ADC converter */
    ioctl(ADC_A, ADC_START, NULL);

    /* wait until SAMPLE 1 is ready */
    while (!ioctl( ADC_A, ADC_GET_STATUS_RDY, 1))
    {
      /* insert the checking of the other events if needed */
    }

    /* early read of ADC SAMPLE 1 (whole scan is not finished yet) */
    w1 = ioctl(ADC_A, ADC_READ_SAMPLE, 1);

    /* wait for end of conversion (whole scan) - CIP = "conversion in progress" */
    while(ioctl(ADC_A, ADC_GET_STATUS_CIP, NULL))
    {
      /* insert the checking of the other events if needed */
    }

    /* now ADC can receive another ADC_START command to begin conversion */

    /* read all samples (8) to buffer */
    ioctl(ADC_A, ADC_READ_ALL_SAMPLES, adc_Buff);

    /* results are in adc_Buff
       set breakpoint to line below and see adc_Buff for results */

    while(1)
    {
        /* keep watchdog clear (if enabled) */
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
    }
}
```

5.10 ADC Driver (MC56F800x)

This section describes the API for the MC56F800x ADC on-chip module. The functionality of the ADC module itself is described in the **56F800x Peripheral Reference Manual**.

5.10.1 Introduction

The MC56F800x devices have two Analog-to-Digital Converter module. The ADC module contains two Status and control registers 1 (ADCSC1A, ADCSC1B), two Date Result registers (ADCRA, ADCRB), Status and Control register 2 (ADCSC2) and one Configuration register (ADCCFG). Each ADC module can be set up to 8bit , 10bit or 12bit conversion mode. The ADC module can be triggered by software when the software trigger is selected writing number of input channel in to the status and control register 1. When the hardware trigger is selected, the ADC module can be triggered by PGA module or by PDB module. The ADC module generates interrupt request, if conversion complete interrupt is enabled.

This section describes the ADC driver software, providing the low level software layer interfacing hardware with software.

5.10.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.10.3.

Table 5-340. ADC Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
ADC0 (ADC_BASE)	0xF060	N/A	N/A	N/A
ADC1 (ADC_BASE)	0xF080	N/A	N/A	N/A

5.10.2.1 API Definition

The following header files are needed in order to use the ADC device driver:

Required Header File(s):

```
#include "qs.h"
#include "adc.h"
```

The following information may be found in the header file *adc.h*.

Public Data Structure(s):

None

5.10.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static ADC module configuration by the driver initialization routine. ADC_*x* should be replaced by ADC_0 for the ADC 0 module and ADC_1 for the ADC 1 module. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-341. ADC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
ADC_x_ADCSC1A_INIT	UWord16	The initial value of the ADC Status and Control Register 1A.
ADC_x_ADCSC1B_INIT	UWord16	The initial value of the ADC Status and Control Register 1B.
ADC_x_ADCSC2_INIT	UWord16	The initial value of the ADC Status and Control Register 2.
ADC_x_ADCCFG_INIT	UWord16	The initial value of the ADC configuration Register.

5.10.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call “changes” the ADC device modes or accesses the ADC register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-342. ADC Driver Arguments - ioctl

pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
cmd	in	Commands found in *adc.h* which are used to modify the ADC module status and control registers. See Table 5-343.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-343. ioctl commands

cmd	param	Return	Description
ADC_INIT	NULL	None	Initializes ADC module by data from the configuration file (appconfig.h).
ADC_SET_CONVERSION_MODE_A	ADC_CONV_SINGLE / ADC_CONV_ CONTINUOUS	None	Selects conversion mode at part A.
ADC_SET_CONVERSION_MODE_B	ADC_CONV_SINGLE / ADC_CONV_ CONTINUOUS	None	Selects conversion mode at part B.
ADC_SET_INPUT_CHANNEL_A	This command accepts always one of the pre-defined values generally formatted as: ADC_channel, (channel=AD0 - AD27/ VREFH/ VREFL/ DEACTIVATE/ VDDCORE/ VDDAREG/ VREF/ TEMPERATURE/ 0_PGA0/ 1_PGA1) Note: The ADC_0_PGA0 channel can be used only at the ADC 0 converter and the ADC_1_PGA1 channel can be used only at the ADC 1 converter.	None	Selects the ADC input channel at part A.

Table 5-343. ioctl commands (Continued)

cmd	param	Return	Description
ADC_SET_INPUT_CHANNEL_B	This command accepts always one of the pre-defined values generally formatted as: ADC_channel, (channel=AD0 - AD27/ VREFH/ VREFL/ DEACTIVATE/ VDDCORE/ VDDAREG/ VREF/ TEMPERATURE/ 0_PGA0/ 1_PGA1) Note: The ADC_0_PGA0 channel can be used only at the ADC 0 converter and the ADC_1_PGA1 channel can be used only at the ADC 1 converter.	None	Selects the ADC input channel at part B.
ADC_TEST_CONVERSION_COMPLETE_A	NULL	UWord16	reads and returns conversion complete flag at part A.
ADC_TEST_CONVERSION_COMPLETE_B	NULL	UWord16	reads and returns conversion complete flag at part B.
ADC_TEST_CONVERSION_ACTIVE	NULL	UWord16	reads and returns conversion active flag.
ADC_SET_CONVERSION_TRIGGER	ADC_TRIGGER_HW / ADC_TRIGGER_SW	None	Selects software or hardware trigger.
ADC_SET_CONVERSION_CLOCK_OUT	ADC_ENABLE / ADC_DISABLE	None	Enables or disables conversion output clock.
ADC_SET_VOLTAGE_REFERENCE	ADC_SOURCE_VREF / ADC_SOURCE_VDDA / ADC_SOURCE_BANDGAP	None	Sets Low or High power mode.
ADC_READ_SAMPLE_A	NULL	UWord16	reads data result register A.
ADC_READ_SAMPLE_B	NULL	UWord16	reads data result register B.
ADC_SET_LOW_POWER_CONF	ADC_HIGH_SPEED / ADC_LOW_SPEED	None	selects low or high power mode.
ADC_SET_DIVIDER	ADC_CLOCK_DIVIDER_1/ ADC_CLOCK_DIVIDER_2/ ADC_CLOCK_DIVIDER_4/ ADC_CLOCK_DIVIDER_8	None	selects clock divider.
ADC_SET_SAMPLE_TIME	ADC_SHORT_TIME / ADC_LONG_TIME	None	Selects long or short sample time.

Table 5-343. ioctl commands (Continued)

cmd	param	Return	Description
ADC_SET_RESOLUTION	ADC_MODE_8BIT / ADC_MODE_10BIT / ADC_MODE_12BIT	None	Selects converson mode.
ADC_SET_CLOCK_INPUT	ADC_CLOCK_SEL_BUS / ADC_CLOCK_SEL_BUS_ DIV2 / ADC_CLOCK_SEL_ ALTCLK / ADC_CLOCK_SEL_ ADATCK	None	Selects ADC clock source.

5.10.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.10.3.1 ADC_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, ADC_INIT, NULL);
```

Arguments:

Table 5-344. ADC_INIT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_INIT* ioctl command executes the initialization function, which initializes all configured ADC registers by the values defined in *appconfig.h*. It is intended for static configuration of the ADC module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_INIT* ioctl command is implemented as a function call. Only necessary ADC registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-316. ADC_INIT

```
ioctl(ADC0, ADC_INIT, NULL);
```

This code initializes the ADC 0 module by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.10.3.2 ADC_SET_CONVERSION_MODE_A - select single/continuous mode

Call(s):

```
void ioctl(const int *pModuleBase,
           ADC_SET_CONVERSION_MODE_A, UWord16 param);
```

Arguments:

Table 5-345. ADC_SET_CONVERSION_MODE_A ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_CONVERSION_SINGLE / ADC_CONVERSION_CONTINUOUS

Description: The *ADC_SET_CONVERSION_MODE_A* ioctl command selects single conversion mode or continuous conversion mode at the ADC module rersult A.

This command writes directly into the Continuous Conversion Enable (ADCO) bit of the ADC Status and Control Register 1A.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_CONVERSION_MODE_A* command is implemented as a macro.

Example 5-317. ADC_SET_CONVERSION_MODE_A

```
ioctl(ADC0, ADC_SET_CONVERSION_MODE_A,ADC_CONVERSION_SINGLE);
```

This code sets the ADC 0 to single conversion at result A.

5.10.3.3 ADC_SET_CONVERSION_MODE_B - select single/continuous mode

Call(s):

```
void ioctl(const int *pModuleBase,
           ADC_SET_CONVERSION_MODE_B, UWord16 param);
```

Arguments:

Table 5-346. ADC_SET_CONVERSION_MODE_B ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_CONVERSION_SINGLE / ADC_CONVERSION_CONTINUOUS

Description: The *ADC_SET_CONVERSION_MODE_B* ioctl command selects single conversion mode or continuous conversion mode at the ADC module rersult A.

This command writes directly into the Continuous Conversion Enable (ADCO) bit of the ADC Status and Control Register 1B.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_CONVERSION_MODE_B* command is implemented as a macro.

Example 5-318. ADC_SET_CONVERSION_MODE_B

```
ioctl(ADC0, ADC_SET_CONVERSION_MODE_B,ADC_CONVERSION_CONTINUOUS);
```

This code sets the ADC 0 to single conversion at result B.

5.10.3.4 ADC_SET_INPUT_CHANNEL_A - select ADC input channel

Call(s):

```
void ioctl(const int *pModuleBase,
           ADC_SET_INPUT_CHANNEL_A, UWord16 param);
```

Arguments:

Table 5-347. ADC_SET_INPUT_CHANNEL_A ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	This command accepts always one of the pre-defined values generally formatted as: ADC_channel, (channel=AD0 - AD27/ VREFH/ VREFL/ DEACTIVATE/ VDDCORE/ VDDAREG/ VREF/ TEMPERATURE/ 0_PGA0/ 1_PGA1) Note: The ADC_0_PGA0 channel can be used only at the ADC 0 converter and the ADC_1_PGA1 channel can be used only at the ADC 1 converter.

Description: The *ADC_SET_INPUT_CHANNEL_A* ioctl command selects input channel. This command can start ADC conversion, when the ADC converter is set to Software Trigger mode. When ADC_DEACTIVATE is used as a parameter, this command disables the ADC module. An internal temperature sensor is connected to the one ADC module input. The temperature input is selected, when ADC_TEMPERATURE is used as a parameter.

This command writes directly into the Input Channel Select (ADCH) bits of the ADC Status and Control Register 1A.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_INPUT_CHANNEL_A* command is implemented as a macro.

Example 5-319. ADC_SET_INPUT_CHANNEL_A

```
ioctl(ADC0, ADC_SET_CONVERSION_TRIGGER, ADC_TRIGGER_SW);
ioctl(ADC0, ADC_SET_INPUT_CHANNEL_A, ADC_AD7);
```

This code sets the ADC0 module to software trigger mode, sets input channel to 7 and initialize ADC conversion.

5.10.3.5 ADC_SET_INPUT_CHANNEL_B - select ADC input channel

Call(s):

```
void ioctl(const int *pModuleBase,
           ADC_SET_INPUT_CHANNEL_B, UWord16 param);
```

Arguments:

Table 5-348. ADC_SET_INPUT_CHANNEL_B ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	This command accepts always one of the pre-defined values generally formatted as: ADC_channel, (channel=AD0 - AD27/ VREFH/ VREFL/ DEACTIVATE/ VDDCORE/ VDDAREG/ VREF/ TEMPERATURE/ 0_PGA0/ 1_PGA1) Note: The ADC_0_PGA0 channel can be used only at the ADC 0 converter and the ADC_1_PGA1 channel can be used only at the ADC 1 converter.

Description: The *ADC_SET_INPUT_CHANNEL_B* ioctl command selects input channel. This command can start ADC conversion, when the ADC converter is set to Software Trigger mode. When ADC_DEACTIVATE is used as a parameter, this command disables the ADC module. An internal temperature sensor is connected to the one ADC module input. The temperature input is selected, when ADC_TEMPERATURE is used as a parameter.

This command writes directly into the Input Channel Select (ADCH) bits of the ADC Status and Control Register 1B.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_INPUT_CHANNEL_B* command is implemented as a macro.

Example 5-320. ADC_SET_INPUT_CHANNEL_B

```
ioctl(ADC0, ADC_SET_INPUT_CHANNEL_B, ADC_0_PGA0);
```

This code sets input channel to the Programable Gain Amplifier output

5.10.3.6 ADC_TEST_CONVERSION_COMPLETE_A - read conversion complete flag

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
                ADC_TEST_CONVERSION_COMPLETE_A, NULL);
```

Arguments:

Table 5-349. ADC_TEST_CONVERSION_COMPLETE_A ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_TEST_CONVERSION_COMPLETE_A* ioctl command reads and returns the Conversion Complete Flag (COCO) bit of the ADC Status and Control Register 1A.

Returns: Non-zero value when the ADC module has finished convesion. Zero when conversion is not completed.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_TEST_CONVERSION_COMPLETE_A* command is implemented as a macro.

Example 5-321. ADC_TEST_CONVERSION_COMPLETE_A

```
if (ioctl(ADC0, ADC_TEST_CONVERSION_COMPLETE_A, NULL))
{
      ...
}
```

This code tests if the ADC0 module has finished coversion.

5.10.3.7 ADC_TEST_CONVERSION_COMPLETE_B - read conversion complete flag

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
                ADC_TEST_CONVERSION_COMPLETE_B, NULL);
```

Arguments:

Table 5-350. ADC_TEST_CONVERSION_COMPLETE_B ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_TEST_CONVERSION_COMPLETE_B* ioctl command reads and returns the Conversion Complete Flag (COCO) bit of the ADC Status and Control Register 1B.

Returns: Non-zero value when the ADC module has finished convesion. Zero when conversion is not completed.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_TEST_CONVERSION_COMPLETE_B* command is implemented as a macro.

Example 5-322. ADC_TEST_CONVERSION_COMPLETE_B

```
if (ioctl(ADC0, ADC_TEST_CONVERSION_COMPLETE_B, NULL))
{
      ...
}
```

This code tests if the ADC0 module has finished coversion.

5.10.3.8 ADC_TEST_CONVERSION_ACTIVE - read conversion active flag

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
                ADC_TEST_CONVERSION_ACTIVE, NULL);
```

Arguments:

Table 5-351. ADC_TEST_CONVERSION_ACTIVE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_TEST_CONVERSION_ACTIVE* ioctl command reads and returns the Conversion Actuve (ADACT) bit of the ADC Status and Control Register 2.

Returns: Non-zero value when the ADC module is performing a conversion. Zero when the ADC module conversion is not in progress.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_TEST_CONVERSION_ACTIVE* command is implemented as a macro.

Example 5-323. ADC_TEST_CONVERSION_ACTIVE

```
if (ioctl(ADC0, ADC_TEST_CONVERSION_ACTIVE, NULL))
{
      ...
}
```

This code tests if the ADC0 module conversion is performing.

5.10.3.9 ADC_SET_TRIGGER_MODE - select trigger mode

Call(s):

```
void ioctl(const int *pModuleBase,ADC_SET_TRIGGER_MODE,
           UWord16 param);
```

Arguments:

Table 5-352. ADC_SET_TRIGGER_MODE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_TRIGGER_SW / ADC_TRIGGER_HW

Description: The *ADC_SET_TRIGGER_MODE* ioctl command sets trigger mode at the ADC module. This command sets hardware trigger mode, when ADC_TRIGGER_HW is used as a parameter. The hardware trigger is generated by Programmable Delay Block (PDB) module or by Programable Gain Amplifier (PGA). When ADC_TRIGGER_SW is used as a parameter, the ADC module can be triggered by selecting ADC input channel.

This command writes directly into the Conversion Trigger Select (ADTRG) bit of the ADC Status and Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_TRIGGER_MODE* command is implemented as a macro.

Example 5-324. ADC_SET_TRIGGER_MODE

```
ioctl(ADC0, ADC_SET_TRIGGER_MODE,ADC_TRIGGER_SW);
ioctl(ADC0, ADC_SET_INPUT_CHANNEL_A,ADC_AD9);
```

This code sets the ADC 0 module to Software Trigger mode and initiates conversion.

5.10.3.10 ADC_SET_CONVERSION_CLOCK_OUT - enables/disables continuous clock output

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CONVERSION_CLOCK_OUT,
           UWord16 param);
```

Arguments:

Table 5-353. ADC_SET_CONVERSION_CLOCK_OUT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_ENABLE/ ADC_DISABLE

Description: The *ADC_SET_CONVERSION_CLOCK_OUT* ioctl command enables or disables the ADC continuous clock output. If the continuous clock output is not required by other on-chip modules (second ADC module, PGA module), is recomended to disable it to conserve power.

This command writes directly into the Enable Continuous Clock output (ECC) bit of the ADC Status and Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_CONVERSION_CLOCK_OUT* command is implemented as a macro.

Example 5-325. ADC_SET_CONVERSION_CLOCK_OUT

```
ioctl(ADC0, ADC_SET_CONVERSION_CLOCK_OUT,ADC_DISABLE);
```

This code disables the continuous clock output.

5.10.3.11 ADC_SET_VOLTAGE_REFERENCE - select volatge reference

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_VOLTAGE_REFERENCE,
           UWord16 param);
```

Arguments:

Table 5-354. ADC_SET_VOLTAGE_REFERENCE ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_SOURCE_VREF/ ADC_SOURCE_VDDA / ADC_SOURCE_BANDGAP

Description: The *ADC_SET_VOLTAGE_REFERENCE* ioctl command selects the ADC voltage reference. This command selects VREF as the voltage reference, when ADC_SOURCE_VREF is used as a parameter. When ADC_SOURCE_VDDA is used as a parameter, the voltage reference is connected to VDDA pins. When ADC_SOURCE_BANDGAP is used as a parameter, the voltage reference is connected to the Bandgap internal reference. The Bandgap internal reference can be filtered by an internal bandgap buffer. The internal bandgap buffer can be enabled by PMC_SET_BANDGAP_BUFFER ioctl command.

This command writes directly into the Voltage Reference Select (REFSEL) bits of the ADC Status and Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_VOLTAGE_REFERENCE* command is implemented as a macro.

Example 5-326. ADC_SET_VOLTAGE_REFERENCE

```
ioctl(PMC, PMC_SET_BANDGAP_BUFFER, PMC_ENABLE);
ioctl(ADC0, ADC_SET_VOLTAGE_REFERENCE, ADC_SOURCE_BANDGAP);
```

This code enables the bandgap reference buffer and selects bandgap reference as the ADC voltage reference.

5.10.3.12 ADC_READ_SAMPLE_A - read sample result A

Call(s):

```
Word16 ioctl(const int *pModuleBase, ADC_READ_SAMPLE_A, NULL);
```

Arguments:

Table 5-355. ADC_READ_SAMPLE_A ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_READ_SAMPLE_A* ioctl command reads a SAMPLE result A. It doesn't check if a new sample is ready, i.e. to find out that a new sample is converted, it is necessary to use other commands.

This command directly reads the ADC Result Register A.

Returns: Sample result.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_SAMPLE_A* command is implemented as a macro.

Example 5-327. ADC_READ_SAMPLE_A

```
Word16 R;
/* read SAMPLE A result */
R = ioctl(ADC0, ADC_READ_SAMPLE_A, NULL);
```

This code reads the ADC0 SAMPLE result A.

5.10.3.13 ADC_READ_SAMPLE_B - read sample result B

Call(s):

```
Word16 ioctl(const int *pModuleBase, ADC_READ_SAMPLE_B, NULL);
```

Arguments:

Table 5-356. ADC_READ_SAMPLE_B ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.

Description: The *ADC_READ_SAMPLE_B* ioctl command reads a SAMPLE result B. It doesn't check if a new sample is ready, i.e. to find out that a new sample is converted, it is necessary to use other commands.

This command directly reads the ADC Result Register B.

Returns: Sample result.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *ADC_READ_SAMPLE_B* command is implemented as a macro.

Example 5-328. ADC_READ_SAMPLE_B

```
Word16 R;
/* read SAMPLE A result */
R = ioctl(ADC0, ADC_READ_SAMPLE_B, NULL);
```

This code reads the ADC0 SAMPLE result B.

5.10.3.14 ADC_SET_DIVIDER - select clock divider

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_DIVIDER,
           UWord16 param);
```

Arguments:

Table 5-357. ADC_SET_DIVIDER ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_CLOCK_DIVIDER_1 / ADC_CLOCK_DIVIDER_2 / ADC_CLOCK_DIVIDER_4 / ADC_CLOCK_DIVIDER_8

Description: The *ADC_SET_DIVIDER* ioctl command selects the divide ratio used by the ADC to generate the internal ADCK.

This command writes directly into the Clock Divide Select (ADIV) bits of the ADC Configuration Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_DIVIDER* command is implemented as a macro.

Example 5-329. ADC_SET_DIVIDER

```
ioctl(ADC0, ADC_SET_DIVIDER, ADC_CLOCK_1);
```

This code selects the divide ratio to 1.

5.10.3.15 ADC_SET_SAMPLE_TIME - select sample time configuration

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_SAMPLE_TIME,
           UWord16 param);
```

Arguments:

Table 5-358. ADC_SET_SAMPLE_TIME ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_SHORT_TIME / ADC_LONG_TIME

Description: The *ADC_SET_SAMPLE_TIME* ioctl command selects the sample time configuration.

This command writes directly into the Long Sample Time Configuration (ADLSMP) bit of the ADC Configuration Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_SAMPLE_TIME* command is implemented as a macro.

Example 5-330. ADC_SET_SAMPLE_TIME

```
ioctl(ADC0, ADC_SET_SAMPLE_TIME, ADC_SHORT_TIME);
```

This code selects short sample time at the ADC 0 module.

5.10.3.16 ADC_SET_RESOLUTION - select conversion mode

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_RESOLUTION,
           UWord16 param);
```

Arguments:

Table 5-359. ADC_SET_RESOLUTION ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_MODE_8BIT / ADC_MODE_10BIT / ADC_MODE_12BIT

Description: The *ADC_SET_RESOLUTION* ioctl command selects the conversion mode. The ADC module can be set up to 12, 10 or 8 bit operation.

This command writes directly into the Conversion Mode Operation (MODE) bits of the ADC Configuration Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_RESOLUTION* command is implemented as a macro.

Example 5-331. ADC_SET_RESOLUTION

```
ioctl(ADC0, ADC_SET_RESOLUTION, ADC_12BIT);
```

This code selects to the 12bits conversion mode.

5.10.3.17 ADC_SET_CLOCK_INPUT - select input clock

Call(s):

```
void ioctl(const int *pModuleBase, ADC_SET_CLOCK_INPUT,
           UWord16 param);
```

Arguments:

Table 5-360. ADC_SET_CLOCK_INPUT ioctl call arguments

*pModuleBase	in	The ADC module identifier. Use ADC0 or ADC1.
param	in	Use one of following values: ADC_CLOCK_SEL_BUS / ADC_CLOCK_SEL_BUS_DIV2 / ADC_CLOCK_SEL_ALTCLK / ADC_CLOCK_SEL_ADACK

Description: The *ADC_SET_CLOCK_INPUT* ioctl command selects the input clock.

This command writes directly into the Input Clock Select (ADICLK) bits of the ADC Configuration Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *ADC_SET_CLOCK_INPUT* command is implemented as a macro.

Example 5-332. ADC_SET_CLOCK_INPUT

```
ioctl(ADC0, ADC_SET_CLOCK_INPUT, ADC_ADACK);
```

This code enables asynchronous clock generator and select its outpus as ADC module clock source.

5.10.4 ADC Driver Applications

There is onee sample applications demonstrating how ADC module can be used in a user application. The supported DEMO boards for ADC sample application are MC56F8006DEMO.

5.10.4.1 adc_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8006DEMO***\adc_demo***

The demo application is using the PWM module to generate two analog signals. The analog signals are connected to the ADC module ANA7 and ANA9 inputs. The ADC module is triggered by Software when AD conversion is completed. Converted analog value is readed and stored in the ADC interrupt rutine.

In FreeMASTER are shown actual values of PWM outputs and actual converted ADC value. PWM output 1 generates a sawtooth signal and PWM output 2 can by set by user. In FreeMASTER it is possible to change reference voltage. Another possibility is possibility to select conversion 8bit/10bit/12bit mode

There are 6 LED diodes on this board. The led diodes PWM0 and PWM1 indicate values of PWM outputs.

LED diode PWM5 indicates that the main loop is working correctly.

LED diode PWM4 indicates that the PWM module is reloading correctly.

LED diode PWM3 indicates that the ADC module is convering correctly.

5.11 PGA Driver

This section describes the API of MC56F800x Programmable Gain Amplifier (PGA) on-chip module. The functionality of the PGA module itself is described in the **56F800x Peripheral Reference Manual**.

5.11.1 Introduction

The MC56F800x devices have one Programmable Gain Amplifier module. The PGA module contains three control registers CNTL0, CNTL1, CNTL2 and one status register STS. The PGA module controls a sample and hold differential amplifier. In the sample and hold differential amplifier can be set up amplifier gain to 1 or 2. The PGA module controls another two differential amplifiers, where amplifier gain can be set up to 1, 2, 3 or 4. The maximum amplifier gain of the PGA module can be set up to 32. See the **56F800x Peripheral Reference Manual** for more details.

This section describes the PGA driver software, providing the low level software layer interfacing hardware with software.

5.11.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.11.3.

Table 5-361. PGA Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
PGA0 (PGA_BASE)	0xF0A0	N/A	N/A	N/A
PGA1 (PGA_BASE)	0xF0C0	N/A	N/A	N/A

5.11.2.1 API Definition

The following header files are needed in order to use the PGA device driver:

Required Header File(s):

```
#include "qs.h"
#include "pga.h"
```

The following information may be found in the header file *pga.h*.

Public Data Structure(s):

None

5.11.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static PGA module configuration by the driver initialization routine. PGA_*x* should be replaced by PGA_0 for the PGA 0 module and PGA_1 for the PGA 1 module.. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-362. PGA Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
PGA_x_CNTL0_INIT	UWord16	The initial value of the PGA Control Register 0
PGA_x_CNTL1_INIT	UWord16	The initial value of the PGA Control Register 1
PGA_x_CNTL2_INIT	UWord16	The initial value of the PGA Control Register 2

5.11.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call “changes” the PGA device modes or accesses the PGA register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-363. PGA Driver Arguments - ioctl

pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1..

Table 5-363. PGA Driver Arguments - ioctl

cmd	in	Commands found in *pga.h* which are used to modify the PGA module status and control registers. See Table 5-364.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-364. ioctl commands

cmd	param	Return	Description
PGA_INIT	NULL	None	Initializes PGA module by data from the configuration file (appconfig.h).
PGA_SET_TRIGGER_MODE	PGA_TRIGGER_SW / PGA_TRIGGER_HW	None	Selects software or hardware trigger.
PGA_SET_GAIN	This command accepts always one of the pre-defined values generally formatted as: PGA_GAIN_*gain* (gain=1X/2X/3X/4X/6x/8X/9X/12X/16X/18X/24X/32X)	None	Selects the PGA module amplifier gain.
PGA_SET_GAIN_SH	PGA_GAIN_SH_1X / PGA_GAIN_SH_2X	None	Selects the PGA sample and hold Gain Stage.
PGA_SET_GAIN_DIFF	PGA_GAIN_DIFF_1X / PGA_GAIN_DIFF_2X / PGA_GAIN_DIFF_3X / PGA_GAIN_DIFF_4X	None	Selects PGA Diff Gain Stage.
PGA_SET_GAIN_DIFF_2	PGA_GAIN_DIFF_2_1X / PGA_GAIN_DIFF_2_2X / PGA_GAIN_DIFF_2_3X / PGA_GAIN_DIFF_2_4X	None	Selects PGA Diff 2 Gain Stage.
PGA_SET_POWER_MODE	PGA_LOW_POWER / PGA_HIGH_POWER	None	Sets Low or High power mode.
PGA_ENABLE_MODULE	NULL	None	Enables the PGA module.
PGA_DISABLE_MODULE	NULL	None	Disables the PGA module.
PGA_SET_SH_BYPASS	PGA_ENABLE / PGA_DISABLE	None	Enables or disables Sample and hold amplifier bypass.

Table 5-364. ioctl commands (Continued)

cmd	param	Return	Description
PGA_SET_CALIBRATION_ MODE	PGA_MISSION_MODE / PGA_OFFSET _CALIBRATION	None	Sets calibration mode.
PGA_SET_CHARGE_PUMP_ DIV	UWord16 (0..7)	None	Sets charge pump divider.
PGA_SET_SW_TRIGGER	NULL	None	Activates SW trigger.
PGA_SET_DIVIDER	PGA_CLOCK_DIVIDER_1/ PGA_CLOCK_DIVIDER_2/ PGA_CLOCK_DIVIDER_4/ PGA_CLOCK_DIVIDER_8	None	Sets clock divider.
PGA_SET_CLK_GS	UWord16 (0..7)	None	Sets number gain clock pulses per conversion.
PGA_TEST_RUNNING	NULL	UWord16	Reads and returns status of conversion complete flag.
PGA_TEST_STARTUP_ COMPLETE	NULL	UWord16	Reads and returns startup complete flag.

5.11.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.11.3.1 PGA_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, PGA_INIT, NULL);
```

Arguments:

Table 5-365. PGA_INIT ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_INIT* ioctl command executes the initialization function, which initializes all configured PGA registers by the values defined in *appconfig.h*. It is intended for static configuration of the PGA module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_INIT* ioctl command is implemented as a function call. Only necessary PGA registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-333. PGA_INIT

```
ioctl(PGA0, PGA_INIT, NULL);
```

This code initializes the PGA 0 module by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.11.3.2 PGA_SET_TRIGGER_MODE - select trigger mode

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_TRIGGER_MODE,
           UWord16 param);
```

Arguments:

Table 5-366. PGA_SET_TRIGGER_MODE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_TRIGGER_SW / PGA_TRIGGER_HW

Description: The *PGA_SET_TRIGGER_MODE* ioctl command sets trigger mode at the PGA module. This command sets hardware trigger mode, when PGA_TRIGGER_HW is used as a parameter. The hardware trigger is generated by Programmable Delay Block (PDB) module. When PGA_TRIGGER_SW is used as a parameter, the Programmable Gain Amplifier can be triggered by PGA_SET_SW_TRIGGER ioctl command.

This command writes directly into the Trigger Mode (TM) bit of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_TRIGGER_MODE* command is implemented as a macro.

Example 5-334. PGA_SET_TRIGGER_MODE

```
ioctl(PGA0, PGA_SET_TRIGGER_MODE,PGA_TRIGGER_SW);
ioctl(PGA0, PGA_SET_SW_TRIGGER,NULL);
```

This code sets the PGA 0 module to Software Trigger mode and initiates conversion.

5.11.3.3 PGA_SET_GAIN - set amplifier gain

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_GAIN, UWord16 param);
```

Arguments:

Table 5-367. PGA_SET_GAIN ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	This command accepts always one of the pre-defined values generally formatted as: PGA_GAIN_*gain* (gain=1X/2X/3X/4X/6x/8X/9X/12X/16X/18X/24X/32X)

Description: The *PGA_SET_GAIN* ioctl command sets an amplifier gain. This command changes the H/S, Diff and Diff2 amplifier gain to set required total amplifier gain.

This command writes directly into the Gain Select (GAINSEL) bits of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_GAIN* command is implemented as a macro.

Example 5-335. PGA_SET_GAIN

```
ioctl(PGA0, PGA_SET_GAIN,PGA_GAIN_32X);
```

This code sets the amplifier gain to 32 at the PGA 0 module.

5.11.3.4 PGA_SET_GAIN_SH - set amplifier gain at Sample and Hold stage

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_GAIN_SH,
           UWord16 param);
```

Arguments:

Table 5-368. PGA_SET_GAIN_SH ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_GAIN_SH_1X / PGA_GAIN_SH_2X

Description: The *PGA_SET_GAIN_SH* ioctl command sets amplifier gain at the Sample and hold stage. The Sample and Hold stage allows to set up amplifier gain to 1 or 2.

This command writes directly into the Gain Select (GAINSEL[0]) bit of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_GAIN_SH* command is implemented as a macro.

Example 5-336. PGA_SET_GAIN_SH

```
ioctl(PGA, PGA_SET_GAIN_SH,PGA_GAIN_1X);
```

This code sets amplifier gain to 1 at the sample and hold stage of PGA 0 module.

5.11.3.5 PGA_SET_GAIN_DIFF - set amplifier gain at differential stage

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_GAIN_DIFF,
           UWord16 param);
```

Arguments:

Table 5-369. PGA_SET_GAIN_DIFF ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_GAIN_DIFF_1X / PGA_GAIN_DIFF_2X / PGA_GAIN_DIFF_3X / PGA_GAIN_DIFF_4X

Description: The *PGA_SET_GAIN_DIFF* ioctl command sets amplifier gain at the differential stage. The differential stage allows to set up amplifier gain to 1, 2, 3 or 4.

This command writes directly into the Gain Select (GAINSEL[2:1]) bits of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_GAIN_DIFF* command is implemented as a macro.

Example 5-337. PGA_SET_GAIN_DIFF

```
ioctl(PGA0, PGA_SET_GAIN_DIFF,PGA_GAIN_3X);
```

This code sets amplifier gain to 3 at the differential stage.

5.11.3.6 PGA_SET_GAIN_DIFF_2 - set amplifier gain at differential-to-single-ended stage

Call(s):

```
void ioctl(const int *pModuleBase, PGA_SET_GAIN_DIFF_2,
           UWord16 param);
```

Arguments:

Table 5-370. PGA_SET_GAIN_DIFF_2 ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_GAIN_DIFF_2_1X / PGA_GAIN_DIFF_2_2X / PGA_GAIN_DIFF_2_3X / PGA_GAIN_DIFF_2_4X

Description: The *PGA_SET_GAIN_DIFF_2* ioctl command sets amplifier gain at the differential-to-single ended stage. The differential-to-single-ended stage allows to set up amplifier gain to 1, 2, 3 or 4.

This command writes directly into the Gain Select (GAINSEL[4:3]) bits of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_GAIN_DIFF_2* command is implemented as a macro.

Example 5-338. PGA_SET_GAIN_DIFF_2

```
ioctl(PGA0, PGA_SET_GAIN_DIFF_2,PGA_GAIN_4X);
```

This code sets amplifier gain to 4 at the differential-to-single-ended stage.

5.11.3.7 PGA_SET_POWER_MODE - select power mode

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_POWER_MODE,
           UWord16 param);
```

Arguments:

Table 5-371. PGA_SET_POWER_MODE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_LOW_POWER / PGA_HIGH_POWER

Description: The *PGA_SET_POWER_MODE* ioctl command selects the power mode. This command sets high power mode when PGA_HIGH_POWER is used as a parameter. With the PGA_LOW_POWER parameter this commands sets low power mode.

This command writes directly into the Power Select (LP) bit of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_POWER_MODE* command is implemented as a macro.

Example 5-339. PGA_SET_POWER_MODE

```
ioctl(PGA0, PGA_SET_POWER_MODE,PGA_HIGH_POWER);
```

This code sets the PGA 0 module to high power mode.

5.11.3.8 PGA_ENABLE_MODULE - enable Programable Gain Amplifier module

Call(s):

```
void ioctl(const int *pModuleBase, PGA_ENABLE_MODULE, NULL);
```

Arguments:

Table 5-372. PGA_ENABLE_MODULE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_ENABLE_MODULE* ioctl command enables the Programable Gain Amplifier module.

This command writes directly into the PGA Enable (EN) bit of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_ENABLE_MODULE* command is implemented as a macro.

Example 5-340. PGA_ENABLE_MODULE

```
ioctl(PGA0, PGA_ENABLE_MODULE, NULL);
while(!ioctl(PGA0,PGA_TEST_STARTUP_COMPLETE,NULL))
{
}
```

This code enables the PGA 0 and wait to module initialization complete.

5.11.3.9 PGA_DISABLE_MODULE - disable Programable Gain Amplifier module

Call(s):

```
void ioctl(const int *pModuleBase, PGA_DISABLE_MODULE, NULL);
```

Arguments:

Table 5-373. PGA_DISABLE_MODULE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_DISABLE_MODULE* ioctl command disables the Programable Gain Amplifier module.

This command writes directly into the PGA Enable (EN) bit of the PGA Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_DISABLE_MODULE* command is implemented as a macro.

Example 5-341. PGA_DISABLE_MODULE

```
ioctl(PGA0, PGA_DISABLE_MODULE, NULL);
```

This code disables the PGA 0 module.

5.11.3.10 PGA_SET_SH_BYPASS - enable/disable bypassat the S/H stage

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_SH_BYPASS,
           UWord16 param);
```

Arguments:

Table 5-374. PGA_SET_SH_BYPASS ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_ENABLE / PGA_DISABLE

Description: The *PGA_SET_SH_BYPASS* ioctl command enables or disables bypass at the Sample and hold stage.

This command writes directly into the S/H Bypass Enable (BP) bit of the PGA Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_SH_BYPASS* command is implemented as a macro.

Example 5-342. PGA_SET_SH_BYPASS

```
ioctl(PGA0, PGA_SET_SH_BYPASS,PGA_DISABLE);
```

This code disables bypass at the S/H stage.

5.11.3.11 PGA_SET_CALIBRATION_MODE - set calibration mode

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_CALIBRATION_MODE,
           UWord16 param);
```

Arguments:

Table 5-375. PGA_SET_CALIBRATION_MODE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_MISSION_MODE / PGA_OFFSET_CALIBRATION

Description: The *PGA_SET_CALIBRATION_MODE* ioctl command sets the calibration mode.

This command writes directly into the Calibration Mode (CALMOD) bits of the PGA Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_CALIBRATION_MODE* command is implemented as a macro.

Example 5-343. PGA_SET_CALIBRATION_MODE

```
ioctl(PGA0, PGA_SET_CALIBRATION_MODE,PGA_DISABLE);
```

This code sets the PGA 0 module to the mission mode.

5.11.3.12 PGA_SET_CHARGE_PUMP_DIV - set charge pump divisor

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_CHARGE_PUMP_DIV,
           UWord16 param);
```

Arguments:

Table 5-376. PGA_SET_CHARGE_PUMP_DIV ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Value to set charge pump divisor

Description: The *PGA_SET_CHARGE_PUMP_DIV* ioctl command sets the charge pump divisor.

This command writes directly into the Charge Pump Divisor (CPD) bits of the PGA Control Register 1.

Returns: None.

Range Issues: UWord16 value must be in range of 0 to 7.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_CHARGE_PUMP_DIV* command is implemented as a macro.

Example 5-344. PGA_SET_CHARGE_PUMP_DIV

```
ioctl(PGA0, PGA_SET_CHARGE_PUMP_DIV,0);
```

This code sets the PGA 0 module to the mission mode.

5.11.3.13 PGA_SET_SW_TRIGGER - activate software trigger

Call(s):

```
void ioctl(const int *pModuleBase, PGA_SET_SW_TRIGGER, NULL);
```

Arguments:

Table 5-377. PGA_SET_SW_TRIGGER ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_SET_SW_TRIGGER* ioctl command activates the software trigger.

This command writes directly into the Software Trigger (SWTRIG) bits of the PGA Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_SW_TRIGGER* command is implemented as a macro.

Example 5-345. PGA_SET_SW_TRIGGER

```
ioctl(PGA, PGA_SET_SW_TRIGGER, NULL);
```

This code activates the PGA 0 module software trigger.

5.11.3.14 PGA_SET_DIVIDER - Select clock divider

Call(s):

```
void ioctl(const int *pModuleBase, PGA_SET_DIVIDER,
           UWord16 param);
```

Arguments:

Table 5-378. PGA_SET_DIVIDER ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Use one of following values: PGA_CLOCK_DIVIDER_1 / PGA_CLOCK_DIVIDER_2 / PGA_CLOCK_DIVIDER_4 / PGA_CLOCK_DIVIDER_8

Description: The *PGA_SET_DIVIDER* ioctl command selects the clock divider.

This command writes directly into the Input Trigger Select (ADIV) bits of the PGA Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_DIVIDER* command is implemented as a macro.

Example 5-346. PGA_SET_DIVIDER

```
ioctl(PGA0, PGA_SET_DIVIDER, PGA_CLOCK_DIVIDER_8);
```

This code sets the PGA 0 clock divider to 8.

5.11.3.15 PGA_SET_CLK_GS - set charge pump divisor

Call(s):

```
void ioctl(const int *pModuleBase,PGA_SET_CLK_GS, UWord16 param);
```

Arguments:

Table 5-379. PGA_SET_CLK_GS ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.
param	in	Number of PGA gain clock pulses per conversion

Description: The *PGA_SET_CLK_GS* ioctl command sets Number of PGA clock pulses per conversion.

This command writes directly into the Number clock GS (NUM_CLK_GS) bits of the PGA Control Register 2.

Returns: None.

Range Issues: UWord16 value must be in range of 0 to 7.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_SET_CLK_GS* command is implemented as a macro.

Example 5-347. PGA_SET_CLK_GS

```
ioctl(PGA0, PGA_SET_CLK_GS,2);
```

This code sets the number of PGA gain clock pulses per conversion to 2.

5.11.3.16 PGA_TEST_RUNNING - read PGA RUN Sequence Underway state

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PGA_TEST_RUNNING, NULL);
```

Arguments:

Table 5-380. PGA_TEST_RUNNING ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_TEST_RUNNING* ioctl command reads and returns the PGA RUN Sequence Underway (RUNNING) bit of the PGA Status Register.

Returns: Non-zero value when the PGA module is performing a conversion. Zero when the PGA state machine is inactive.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_TEST_RUNNING* command is implemented as a macro.

Example 5-348. PGA_TEST_RUNNING

```
if (ioctl(PGA, PGA_TEST_RUNNING, NULL))
{
      ...
}
```

This code tests if the PGA module is performing a coversion.

5.11.3.17 PGA_TEST_STARTUP_COMPLETE - read startup complete flag

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PGA_TEST_STARTUP_COMPLETE,
      NULL);
```

Arguments:

Table 5-381. PGA_TEST_STARTUP_COMPLETE ioctl call arguments

*pModuleBase	in	The PGA module identifier. Use PGA0 or PGA1.

Description: The *PGA_TEST_STARTUP_COMPLETE* ioctl command reads and returns the Startup Complete (STCOMP) bit of the PGA Status Register.

Returns: Non-zero value when the PGA module is enabled and has completed its startup sequence. Zero when the PGA module is starting up.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PGA_TEST_STARTUP_COMPLETE* command is implemented as a macro.

Example 5-349. PGA_TEST_STARTUP_COMPLETE

```
if (ioctl(PGA, PGA_TEST_STARTUP_COMPLETE, NULL))
{
      ...
}
```

This code tests if the PGA module is enabled and has completed its startup sequence.

5.11.4 PGA Driver Applications

There is onee sample applications demonstrating how PGA module can be used in a user application. The supported DEMO boards for PGA sample application are MC56F8006DEMO.

5.11.4.1 pga_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8006DEMO***\pga_demo***

The demo application is using the PWM module to generate two analog signals. The analog signals are connected to the PGA module input.The PGA module is triggered by Software every 8 PWM cycles. Converted analog value is readed and stored in ADC interrupt rutine.

In FreeMASTER are shown actual values of PWM outputs and actual converted ADC values. PWM output 1 generates a sawtooth signal and PWM output 2 generates a reverse sawtooth signal. In FreeMASTER it is possible to change amplifier gain. Another possibility is automatic amplifier gain - amplifier gain depends on actual input voltage value.

There are 6 LED diodes on this board. The led diodes PWM0 and PWM1 indicate values of PWM outputs.

LED diode PWM5 indicates that the main loop is working correctly.

LED diode PWM4 indicates that the PWM module is reloading correctly.

LED diode PWM3 indicates that the ADC module is convering correctly.

5.12 PDB Driver

This section describes the API of MC56F800x Programmable Delay Block (PDB) on-chip module. The functionality of the PDB module itself is described in the **56F800x Peripheral Reference Manual**.

5.12.1 Introduction

The MC56F800x devices have one Programable Delay Block module. The PDB module contains a status and control register, a 16-bit up counter, 16-bit modulo register and two 16-bit delay registers DELAY A and DELAY B. The modulo and control registers are read/writable. The counter is read only. The modulo register is loaded with a value to count to and the pre-scaler is set to determine the counting rate. When enabled, the counter counts up to the modulo value, then it resets to $0000. When the counter counts up to the Delay value, PDB mode generates Trigger A or B output pulse. See the **56F800x Peripheral Reference Manual** for more details.

This section describes the PDB driver software, providing the low level software layer interfacing hardware with software.

5.12.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.12.3.

Table 5-382. PDB Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
PDB (PDB_BASE)	0xF300	N/A	N/A	N/A

5.12.2.1 API Definition

The following header files are needed in order to use the PDB device driver:

Required Header File(s):

```
#include "qs.h"
#include "pdb.h"
```

The following information may be found in the header file *pdb.h*.

Public Data Structure(s):

None

5.12.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static PDB module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-383. PDB Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
PDB_SCR_INIT	UWord16	The initial value of the PDB Status and Control Register
PDB_DELAYA_INIT	UWord16	The initial value of the PDB DelayA Register
PDB_DELAYB_INIT	UWord16	The initial value of the PDB DelayB Register
PDB_MOD_INIT	UWord16	The initial value of the PDB Counter Modulus Register

5.12.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the PDB device modes or accesses the PDB register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-384. PDB Driver Arguments - ioctl

pModuleBase	in	PDB module identifier. Use PDB.

Table 5-384. PDB Driver Arguments - ioctl

cmd	in	Commands found in *pdb.h* which are used to modify the PDB module status and control registers. See Table 5-385.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-385. ioctl commands

cmd	param	Return	Description
PDB_INIT	NULL	None	Initializes PDB module by data from the configuration file (appconfig.h).
PDB_SET_PRESCALER	This command accepts always one of the pre-defined values generally formatted as: PDB_CLOCK_*DIVIDER_prescaler* (prescaler= 1/2/4/8/16/32/64/128)	None	Sets PDB counter prescaler.
PDB_SET_TRIGGER_A_OUT	PDB_TRIGGER_A_BYPASS / PDB_TRIGGER_A_DELAY_A / PDB_TRIGGER_A_DELAY_AB / PDB_TRIGGER_A_PULSE_OUT	None	Selects PDB trigger A output.
PDB_SET_TRIGGER_B_OUT	PDB_TRIGGER_B_BYPASS / PDB_TRIGGER_B_DELAY_B / PDB_TRIGGER_B_DELAY_AB / PDB_TRIGGER_B_PULSE_OUT	None	Selects PDB trigger B output.
PDB_SET_CONTINUOUS_MODE	PDB_ENABLE/ PDB_DISABLE	None	Enables/disables continuous mode.
PDB_SET_SW_TRIGGER	NULL	None	Activates SW trigger.

Table 5-385. ioctl commands (Continued)

cmd	param	Return	Description
PDB_SET_INPUT_TRIGGER	PDB_TRIG_CMP0_OUT / PDB_TRIG_CMP1_OUT / PDB_TRIG_CMP2_OUT / PDB_TRIG_PWM_SYNC / PDB_TRIG_EXTERNAL / PDB_TRIG_T0 / PDB_TRIG_T1 / PDB_TRIG_SOFTWARE	None	Selects input trigger source.
PDB_SET_TRIGGER_A_ ENABLE	NULL	None	Enables PDB Trigger A.
PDB_SET_TRIGGER_A_ DISABLE	NULL	None	Disables PDB Trigger A.
PDB_SET_TRIGGER_B_ ENABLE	NULL	None	Enables PDB Trigger B.
PDB_SET_TRIGGER_B_ DISABLE	NULL	None	Disables PDB Trigger B.
PDB_WRITE_DELAYA	UWord16	None	Sets PDB DELAYA value.
PDB_WRITE_DELAYB	UWord16	None	Sets PDB DELAYB value.
PDB_WRITE_MOD	UWord16	None	Sets PDB modulo value.
PDB_READ_COUNTER_REG	NULL	UWord16	Reads PDB immediate counter value.

5.12.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.12.3.1 PDB_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, PDB_INIT, NULL);
```

Arguments:

Table 5-386. PDB_INIT ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_INIT* ioctl command executes the initialization function, which initializes all configured PDB registers by the values defined in *appconfig.h*. It is intended for static configuration of the PDB module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_INIT* ioctl command is implemented as a function call. Only necessary PDB registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-350. PDB_INIT

```
ioctl(PDB, PDB_INIT, NULL);
```

This code initializes the PDB module by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.12.3.2 PDB_SET_PRESCALER - select counter prescaler

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-387. PDB_SET_PRESCALER ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	This command accepts always one of the pre-defined values generally formatted as: PDB_CLOCK_*DIVIDER_prescaler* (prescaler= 1/2/4/8/16/32/64/128)

Description: The *PDB_SET_PRESCALER* ioctl command sets the clock prescaler.

This command writes directly into the Clock Prescaler Select (PRESCALER) bits of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_PRESCALER* command is implemented as a macro.

Example 5-351. PDB_SET_PRESCALER

```
ioctl(PDB, PDB_SET_PRESCALER,PDB_CLOCK_DIVIDER_16);
```

This code sets clock source prescaler to 16.

5.12.3.3 PDB_SET_TRIGGER_A_OUT - select trigger A output

Call(s):

```
void ioctl(const int *pModuleBase,PDB_SET_TRIGGER_A_OUT,
           UWord16 param);
```

Arguments:

Table 5-388. PDB_SET_TRIGGER_A_OUT ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	Use one of following values: PDB_TRIGGER_A_BYPASS / PDB_TRIGGER_A_DELAY_A / PDB_TRIGGER_A_DELAY_AB/ PDB_TRIGGER_A_PULSE_OUT

Description: The *PDB_SET_TRIGGER_A_OUT* ioctl command sets the trigger A output mode. This command bypasses the Trigger A Output of the PDB module when PDB_TRIGGER_A_BYPASS is used as a parameter. With the PDB_TRIGGER_A_DELAY_A parameter the Trigger A Output generates pulse when the counter value reached the DELAY_A value. The command parameter PDB_TRIGGER_A_DELAY_AB is used in ping-pong operation to generate Trigger A pulse when the counter value reaching DELAY A and DELAY B value. With the PDB_TRIGGER_A_PULSE_OUT parameter this command sets the Trigger A output to Trigger Pulsed Output Operation.

This command writes directly into the Trigger A Output Selects (AOS) bits of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_A_OUT* command is implemented as a macro.

Example 5-352. PDB_SET_TRIGGER_A_OUT

```
ioctl(PDB, PDB_SET_TRIGGER_A_OUT,PDB_TRIGGER_A_PULSE_OUT);
```

This code sets the Trigger A Output to the Trigger Pulsed Output Operation.

5.12.3.4 PDB_SET_TRIGGER_B_OUT - select trigger B output

Call(s):

```
void ioctl(const int *pModuleBase,PDB_SET_TRIGGER_B_OUT,
           UWord16 param);
```

Arguments:

Table 5-389. PDB_SET_TRIGGER_B_OUT ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	Use one of following values: PDB_TRIGGER_B_BYPASS / PDB_TRIGGER_B_DELAY_B / PDB_TRIGGER_B_DELAY_AB/ PDB_TRIGGER_B_PULSE_OUT

Description: The *PDB_SET_TRIGGER_B_OUT* ioctl command sets the trigger Boutput mode. This command bypasses the Trigger B Output of the PDB module when PDB_TRIGGER_B_BYPASS is used as a parameter. With the PDB_TRIGGER_B_DELAY_B parameter the Trigger A Output generates pulse when the counter value reaching the DELAY_B value. The command parameter PDB_TRIGGER_B_DELAY_AB is used in ping-pong operation to generate Trigger B pulse when the counter value reaching DELAY A and DELAY B value. With the PDB_TRIGGER_B_PULSE_OUT parameter this command sets the Trigger B output to Trigger Pulsed Output Operation.

This command writes directly into the Trigger B Output Selects (BOS) bits of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_B_OUT* command is implemented as a macro.

Example 5-353. PDB_SET_TRIGGER_B_OUT

```
ioctl(PDB, PDB_SET_TRIGGER_B_OUT,PDB_TRIGGER_B_DELAY_AB);
```

This code sets the Trigger B Output to the Ping-Pong Operation.

5.12.3.5 PDB_SET_CONTINUOUS_MODE - enable/disable continuous mode

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_CONTINUOUS_MODE,
           UWord16 param);
```

Arguments:

Table 5-390. PDB_SET_CONTINUOUS_MODE ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	Use one of following values: PDB_ENABLE / PDB_DISABLE

Description: The *PDB_SET_CONTINUOUS_MODE* ioctl command enables or disables the continuous mode.

This command writes directly into the Continuous Mode Enable (COUNT) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_CONTINUOUS_MODE* command is implemented as a macro.

Example 5-354. PDB_SET_CONTINUOUS_MODE

```
ioctl(PDB, PDB_SET_CONTINUOUS_MODE, PDB_ENABLE);
```

This code enables continuous mode.

5.12.3.6 PDB_SET_SW_TRIGGER - activate software trigger

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_SW_TRIGGER, NULL);
```

Arguments:

Table 5-391. PDB_SET_SW_TRIGGER ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_SET_SW_TRIGGER* ioctl command activates the software trigger.

This command writes directly into the Software Trigger (SWTRIG) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_SW_TRIGGER* command is implemented as a macro.

Example 5-355. PDB_SET_SW_TRIGGER

```
ioctl(PDB, PDB_SET_SW_TRIGGER, NULL);
```

This code activates software trigger.

5.12.3.7 PDB_SET_INPUT_TRIGGER - Select input trigger source

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_INPUT_TRIGGER,
           UWord16 param);
```

Arguments:

Table 5-392. PDB_SET_INPUT_TRIGGER ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	Use one of following values: PDB_TRIG_CMP0_OUT / PDB_TRIG_CMP1_OUT / PDB_TRIG_CMP2_OUT / PDB_TRIG_PWM_SYNC / PDB_TRIG_EXTERNAL / PDB_TRIG_T0 / PDB_TRIG_T1 / PDB_TRIG_SOFTWARE

Description: The *PDB_SET_INPUT_TRIGGER* ioctl command selects the PDB input trigger source. This command sets High speed comparator trigger, when PDB_TRIG_CMP0_OUT, PDB_TRIG_CMP1_OUT or PDB_TRIG_CMP2_OUT is used as a parameter. With PDB_TRIG_PWM_SYNC parameter command sets PWM SYNC as a trigger source. The PDB module is triggered by external source, when PDB_TRIG_EXTERNAL is used as a command parameter. This command sets a Dual Timer as trigger source, when PDB_TRIG_T0 or PDB_TRIG_T1 is used as a parameter. The PDB module can be triggered by software when command parameter is PDB_TRIG_SOFTWARE.

This command writes directly into the Input Trigger Select (TRIGSEL) bits of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_INPUT_TRIGGER* command is implemented as a macro.

Example 5-356. PDB_SET_INPUT_TRIGGER

```
ioctl(PDB, PDB_SET_INPUT_TRIGGER, PDB_TRIG_T0);
```

This code sets timer 0 as the trigger source.

5.12.3.8 PDB_SET_TRIGGER_A_ENABLE - enable trigger A output

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_TRIGGER_A_ENABLE,
     NULL);
```

Arguments:

Table 5-393. PDB_SET_TRIGGER_A_ENABLE ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_SET_TRIGGER_A_ENABLE* ioctl command enables the PDB Trigger A output. The PDB Trigger A is generated when PDB counter value reached the DELAY A value.

This command writes directly into the PDB Trigger A Enable (ENA) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_A_ENABLE* command is implemented as a macro.

Example 5-357. PDB_SET_TRIGGER_A_ENABLE

```
ioctl(PDB, PDB_SET_TRIGGER_A_ENABLE, NULL);
```

This code enables the PDB Trigger A output.

5.12.3.9 PDB_SET_TRIGGER_A_DISABLE - disable trigger A output

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_TRIGGER_A_DISABLE,
     NULL);
```

Arguments:

Table 5-394. PDB_SET_TRIGGER_A_DISABLE ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_SET_TRIGGER_A_DISABLE* ioctl command disables the PDB Trigger A output.

This command writes directly into the PDB Trigger A Enable (ENA) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_A_DISABLE* command is implemented as a macro.

Example 5-358. PDB_SET_TRIGGER_A_DISABLE

```
ioctl(PDB, PDB_SET_TRIGGER_A_DISABLE, NULL);
```

This code disables the PDB Trigger A output.

5.12.3.10 PDB_SET_TRIGGER_B_ENABLE - enable trigger B output

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_TRIGGER_B_ENABLE,
     NULL);
```

Arguments:

Table 5-395. PDB_SET_TRIGGER_B_ENABLE ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_SET_TRIGGER_B_ENABLE* ioctl command enables the PDB Trigger B output. The PDB Trigger B is generated when PDB counter value reached the DELAY B value.

This command writes directly into the PDB Trigger B Enable (ENB) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_B_ENABLE* command is implemented as a macro.

Example 5-359. PDB_SET_TRIGGER_B_ENABLE

```
ioctl(PDB, PDB_SET_TRIGGER_B_ENABLE, NULL);
```

This code enables the PDB Trigger B output.

5.12.3.11 PDB_SET_TRIGGER_B_DISABLE - disable trigger B output

Call(s):

```
void ioctl(const int *pModuleBase, PDB_SET_TRIGGER_B_DISABLE,
     NULL);
```

Arguments:

Table 5-396. PDB_SET_TRIGGER_B_DISABLE ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_SET_TRIGGER_B_DISABLE* ioctl command disables the PDB Trigger B output.

This command writes directly into the PDB Trigger B Enable (ENB) bit of the PDB Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_SET_TRIGGER_B_DISABLE* command is implemented as a macro.

Example 5-360. PDB_SET_TRIGGER_B_DISABLE

```
ioctl(PDB, PDB_SET_TRIGGER_B_DISABLE, NULL);
```

This code disables the PDB Trigger B output.

5.12.3.12 PDB_WRITE_DELAYA - set PDB Delay A value

Call(s):

```
void ioctl(const int *pModuleBase, PDB_WRITE_DELAYA,
     UWord16 param);
```

Arguments:

Table 5-397. PDB_WRITE_DELAYA ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	UWord16 DELAY A value.

Description: The *PDB_WRITE_DELAYA* ioctl command writes the DELAY A register, which presents the delay between the input PDB trigger and trigger A output. When counter reaches the DELAY A value, the PDB module generates Trigger A pulse.

This command writes directly into the PDB DELAY A-Value Register (DELAYA).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_WRITE_DELAYA* command is implemented as a macro.

Example 5-361. PDB_WRITE_DELAYA

```
ioctl(PDB, PDB_WRITE_DELAYA, 200);
```

This code configures PDB module to generate Trigger A pulse after 200 counter clocks.

5.12.3.13 PDB_WRITE_DELAYA - set PDB Delay B value

Call(s):

```
void ioctl(const int *pModuleBase, PDB_WRITE_DELAYA,
     UWord16 param);
```

Arguments:

Table 5-398. PDB_WRITE_DELAYA ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	UWord16 DELAY B value.

Description: The *PDB_WRITE_DELAYA* ioctl command writes the DELAY B register, which presents the delay between the input PDB trigger and trigger B output. When counter reaches the DELAY B value, the PDB module generates Trigger B pulse.

This command writes directly into the PDB DELAY B-Value Register (DELAYB).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_WRITE_DELAYA* command is implemented as a macro.

Example 5-362. PDB_WRITE_DELAYA

```
ioctl(PDB, PDB_WRITE_DELAYA, 400);
```

This code configures PDB module to generate Trigger B pulse after 400 counter clocks.

5.12.3.14 PDB_WRITE_MOD - set PDB counter terminal value

Call(s):

```
void ioctl(const int *pModuleBase, PDB_WRITE_MOD, UWord16 param);
```

Arguments:

Table 5-399. PDB_WRITE_MOD ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.
param	in	UWord16 modulo value.

Description: The *PDB_WRITE_MOD* ioctl command writes the counter modulo register, which presents the terminal value of the PDB counter. When counter reaches the modulo value, it is wrapped to zero.

This command writes directly into the Modulo-Value Register (MOD).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_WRITE_MOD* command is implemented as a macro.

Example 5-363. PDB_WRITE_MOD

```
ioctl(PDB, PDB_WRITE_MOD, 30000);
```

This code configures PDB module to end cycle after 30000 clocks.

5.12.3.15 PDB_READ_COUNTER_REG - read PDB counter value

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PDB_READ_COUNTER_REG,NULL);
```

Arguments:

Table 5-400. PDB_READ_COUNTER_REG ioctl call arguments

*pModuleBase	in	The PDB module identifier. Use PDB.

Description: The *PDB_READ_COUNTER_REG* ioctl command reads and returns the PDB Counter register content.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PDB_READ_COUNTER_REG* command is implemented as a macro.

Example 5-364. PDB_READ_COUNTER_REG

```
UWord16 count;

count = ioctl(PDB, PDB_READ_COUNTER_REG, NULL);
```

This code reads the immediate counter value of PDB.

5.12.4 PDB Driver Applications

There is onee sample applications demonstrating how PDB module can be used in a user application. The supported DEMO boards for PDB sample application are MC56F8006DEMO.

5.12.4.1 PDB_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8006DEMO***pdb_demo***

This application demonstrates a Programable Delay Block with a ADC module in ping-pong mode. The demo application is using the PWM module to generate one analog signal.

In FreeMASTER is shown actual value of PWM output, where is a triangle signal. The PDB module is triggered by software when the PWM output reach minimum or maximum its value. In second time graph is shown half value of the internal PDB counter (sawtooth signal) and two sampled analog signal. On both analog inputs is same signal from the PWM module. Sampled analog signals have different values, because both are sampled in different time - sample time depend on values delayA and delayB.

There are 6 LED diodes on this board.

The led diode PWM0 indicate value of PWM output.

LED diode PWM5 indicates that the main loop is working correctly.

LED diode PWM4 indicates that the PWM module is reloading correctly.

LED diode PWM3 indicates that in the PDB module is delayA and delayB working correctly.

5.13 Quadrature Decoder Driver

This section describes the API for the MC56F83xx Quadrature Decoder on-chip module. The functionality of the Quadrature Decoder module itself is described in the **MC56F8300 Peripheral User Manual**.

5.13.1 Introduction

The MC56F83xx has up to two Quadrature Decoders, Quadrature Decoder #0 and Quadrature Decoder #1. Each Quadrature Decoder module has four input signals: PHASEA, PHASEB, INDEX, and HOME.

This section describes the Quadrature Decoder driver software providing the lowest level software layer, interfacing hardware to software.

5.13.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.13.3.

Table 5-401. Quadrature Decoder Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F832x	MC56F83xx
Decoder #0 (Decoder0_BASE)	N/A	N/A	0xF180	0xF180
Decoder #1 (Decoder1_BASE)	N/A	N/A	N/A	0xF190

5.13.2.1 API Definition

The following header files are needed in order to use the Quadrature Decoder device driver:

Required Header File(s):

```
#include "qs.h"
#include "decoder.h"
```

The following information may be found in the header file *decoder.h*.

Public Data Structure(s):

```
typedef union
{
      struct
      {
            UWord16 LSBpart;
            Word16 MSBpart;
      }RegParts;
      Word32 Reg32bit;
}decoder_uReg32bit;

typedef struct
{
      union { Word16 PositionDifferenceHoldReg;
              Word16 posdh; };
      union { Word16 RevolutionHoldReg;
              Word16 revh; };
      union { decoder_uReg32bit PositionHoldReg;
              Word32 posh; };
}decoder_sState;

typedef struct
{
      /* public members */
      UWord16 EncPulses;
      UWord16 RevolutionScale;
      /* internal-private members */
      Int16 scaleDiffPosCoef;
      UInt16 scalePosCoef;
      Int16 normDiffPosCoef;
      Int16 normPosCoef;
}decoder_sEncScale;

typedef struct
{
      UWord16    Index : 1;
      UWord16    PhaseB : 1;
      UWord16    PhaseA : 1;
      UWord16    Reserved : 13;
}decoder_sEncSignals;

typedef union
{
      decoder_sEncSignals   EncSignals;
      UWord16               Value;
}decoder_uEncSignals;
```

Members:

Table 5-402. decoder_sState Data Structure Members

Member	Type	Description
posdh *or PositionDifferenceHoldReg*	Word16 *Word16*	A data value representing the content of the Position Difference Hold Register.
revh *or RevolutionHoldReg*	Word16 *Word16*	A data value representing the content of the Revolution Hold Register.
posh *or PositionHoldReg*	Word32 *uReg32bit*	A data value representing the content of the Lower and Upper Position Hold Register.

Table 5-403. decoder_sEncScale Data Structure Members

Member	Type	Description
EncPulses	UWord16	A data value representing the number of pulses per revolution of the Encoder device.
RevolutionScale	UWord16	A data value representing the number of revolutions to be reflected by the 16 bit register full range. RevolutionScale = 0 represents a range +/- PI RevolutionScale =1 represents a range +/- 2PI RevolutionScale =2 represents a range +/- 4PI ... Note: the maximum RevolutionScale value is determined by the following expression: 32767 > ((EncPulses * 4) * RevolutionScale) When RevolutionScale is equal to 0, then the following expression must be valid: 32767 > (EncPulses * 2) The RevolutionScale has meaning only for DEC_GET_SCALED_POSITION_DIFFERENCE command.
scaleDiffPosCoef	Int16	Precalculated internal scale coefficient - user should not access it
scalePosCoef	UInt16	Precalculated internal scale coefficient - user should not access it
normDiffPosCoef	Int16	Precalculated internal scale coefficient - user should not access it
normPosCoef	Int16	Precalculated internal scale coefficient - user should not access it

Table 5-404. decoder_sEncSignals Data Structure Members

Member	Type	Description
Index	UWord16 : 1	A bit reflecting the state of the Encoder Index Signal.
PhaseB	UWord16 : 1	A bit reflecting the state of the Encoder PhaseB Signal.

Table 5-404. decoder_sEncSignals Data Structure Members (Continued)

Member	Type	Description
PhaseA	UWord16 : 1	A bit reflecting the state of the Encoder PhaseA Signal.
Reserved	UWord16 : 13	Unused

5.13.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static Quadrature Decoder module configuration by the driver initialization routine. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-400 for more details.

Table 5-405. Quadrature Decoder Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
DEC_x_DECCR_INIT	UWord16	Represents contents of the Decoder Control Register
DEC_x_WTR_INIT	UWord16	Represents contents of the Decoder Watchdog Time-out Register
DEC_x_FIR_INIT	UWord16	Represents contents of the Decoder Filter Interval Register
DEC_x_UIR_INIT	UWord16	Represents contents of the Decoder Upper Initialization Register
DEC_x_LIR_INIT	UWord16	Represents contents of the Decoder Lower Initialization Register

Note - x specifies the used Quadrature Decoder module. It can be either 0 or 1.

5.13.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void Cmd, void *pParam);
void ioctl(const int *pModuleBase, void Cmd, void *pParam);
UWord16 ioctl(const int *pModuleBase, void Cmd, UWord16 param);
void ioctl(const int *pModuleBase, void Cmd, UWord16 param);
```

Description: The *ioctl* call "modifies" the Quadrature Decoder device modes or accesses the Quadrature Decoder register(s).

Arguments:

Table 5-406. Quadrature Decoder Driver Arguments - ioctl

pModuleBase	in	Quadrature Decoder module identifier. Use DEC_0 or DEC_1.
Cmd	in	Commands found in *decoder.h* which are used to modify the Quadrature Decoder module status and control registers. See Table 5-407.
pParam, param	in, out, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-407. ioctl commands

Cmd	pParam, param	Return	Description
DEC_INIT	NULL	None	Initializes Quadrature Decoder module by data from configuration file (appconfig.h).
DEC_INT_ENABLE	DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT	None	Enables the selected Quadrature Decoder interrupts.
DEC_INT_DISABLE	DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT	None	Disables the selected Quadrature Decoder interrupts.
DEC_HOME_INT *DEC_INDEX_PULSE_INT* *DEC_WATCHDOG_INT*	*DEC_ENABLE / DEC_DISABLE*	*None*	*Interrupt enable or disable for individual interrupt sources. The same operation possible with DEC_INT_ENABLE / DISABLE*

Table 5-407. ioctl commands (Continued)

Cmd	pParam, param	Return	Description
DEC_INT_REQUEST_CLEAR	DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT	None	Clears the selected Quadrature Decoder interrupt flag(s).
DEC_CLEAR_HOME_INT_REQUEST DEC_CLEAR_INDEX_PULSE_INT_REQUEST DEC_CLEAR_WATCHDOG_INT_REQUEST	*NULL*	*None*	*Clears the individual interrupt request flags. The same operation possible with DEC_INT_REQUEST_CLEAR*
DEC_HOME_TRIGGERED_INIT	DEC_ENABLE / DEC_DISABLE	None	Enables or disables the position counter to be initialized by the HOME signal.
DEC_HOME_EDGE	DEC_NEGATIVE / DEC_POSITIVE	None	Sets the negative or positive edge of the HOME signal to initialize the position counter.
DEC_SOFTWARE_TRIGGERED_INIT	NULL	None	Initializes the position counter by value from the initialization register.
DEC_DIRECTION_COUNTING_ENABLE	DEC_REVERSE / DEC_NORMAL	None	Reverses the interpretation of the quadrature signal to change the direction of count.
DEC_SINGLE_PHASE_COUNT	DEC_ENABLE / DEC_DISABLE	None	If DEC_ENABLE, then the quadrature decoder logic is bypassed (PHASEA is used a single phase pulse stream, PHASEB is ignored).
DEC_INDEX_TRIGGERED_INIT	DEC_ENABLE / DEC_DISABLE	None	Enables or disables the position counter to be initialized by the INDEX pulse.
DEC_INDEX_EDGE	DEC_NEGATIVE / DEC_POSITIVE	None	Sets the negative or positive edge of the INDEX pulse to initialize the position counter.
DEC_WATCHDOG	DEC_ENABLE / DEC_DISABLE	None	Watchdog timer enable or disable.
DEC_SWITCH_MATRIX	DEC_MODE_1 / DEC_MODE_2 / DEC_MODE_3	None	Selects the switch matrix mode that connects the input to the Timer module.
DEC_WRITE_FILTER	UWord16	None	Sets the Filter Interval Register with the desired value.
DEC_WRITE_WATCHDOG_TIMEOUT	UWord16	None	Sets the Watchdog Time-out Register with the desired value.
DEC_READ_POSITION_DIFFERENCE	NULL	Word16	Returns the content of the Position Difference Counter Register. +

Table 5-407. ioctl commands (Continued)

Cmd	pParam, param	Return	Description
DEC_READ_REVOLUTION	NULL	Word16	Returns the content of the Revolution Counter Register. +
DEC_WRITE_REVOLUTION	Word16	None	Sets the Revolution Counter by the desired value.
DEC_READ_POSITION	uReg32bit*	None	Returns the content of the Upper and Lower Position Counter Register. +
DEC_WRITE_POSITION	Word32	None	Sets the Position Counter Register to the desired starting value.
DEC_WRITE_INIT_STATE	Word32	None	Sets the Initialization Position Counter Register (Upper and Lower) with the desired value.
DEC_READ_MONITOR_REG	NULL	UWord16	Returns the content of the Input Monitor Register.
DEC_GET_RAW_ENCSIGNALS	NULL	decoder_uEncSignals.Value	Returns the raw version of INDEX, PHASEB and PHASEA encoder signals.
DEC_GET_FILTERED_ENCSIGNALS	NULL	decoder_uEncSignals.Value	Returns the filtered version of INDEX, PHASEB and PHASEA encoder signals.
DEC_READ_HOLD_DATA_REGS	decoder_sState*	None	Holds the Position, Position Difference and the Revolution counters and copies such state to the structure members. +
DEC_READ_CONTROL_REG	NULL	UWord16	Returns the content of the Control Register.
DEC_CALCULATE_SCALE_COEF	decoder_sEncScale*	None	Calculates the scaling coefficients needed for the correct functionality of DEC_GET_SCALED_POSITION and DEC_GET_SCALED_POSITION_DIFFERENCE commands (i.e. this command must be executed before the execution of the above mentioned commands).

Table 5-407. ioctl commands (Continued)

Cmd	pParam, param	Return	Description
DEC_GET_SCALED_POSITION	decoder_sEncScale*	Word32	Calculates an absolute position. It returns a 32bit value where the MSB part represents the number of revolutions (equals to the content of the Revolution Register) while the LSB part represents the portion of the current revolution scaled into the 16bit unsigned data range. The DEC_CALCULATE_SCALE_COEF command must be executed prior to this command. Note: the correct functionality requires to fill the Initialization Register by value 0x00000000 and to enable initialization of the Position Counter by the INDEX signal (DEC_WRITE_INIT_STATE and DEC_INDEX_TRIGGERED_INIT commands) during the Quadrature Decoder init phase. +
DEC_GET_SCALED_POSITION_DIFFERENCE	decoder_sEncScale*	Word16	Returns the scaled relative position (difference). The 16bit signed value represents a range specified by the RevolutionScale. The DEC_CALCULATE_SCALE_COEF command must be executed prior to this command. Note: this command recalculates (scales) the content of the Position Difference Counter Register which is automatically cleared when Position Register is read. +

+ Note - Causes to copy all Quadrature Decoder counter registers' contents to their corresponding hold registers.

5.13.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the ioctl commands usage.

5.13.3.1 DEC_INIT - initialize Quadrature Decoder module

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INIT, NULL);
```

Arguments:

Table 5-408. DEC_INIT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_INIT* ioctl command is used to initialize the selected Quadrature Decoder module. Initialization consists of writing the initialization values to the following Decoder registers: Decoder Control Register (DECCR), Watchdog Timeout Register (WTR), Filter Interval Register (FIR), Upper Initialization Register (UIR) and Lower Initialization Register (LIR). Initialization values (configuration items) for each register are taken from *appconfig.h*, where each configuration item corresponds to one Quadrature Decoder register. If no initialization values is defined in *appconfig.h,* then no initialization is performed. It is not necessary to define the initialization values for all Quadrature Decoder registers in *appconfig.h*, define only those which are needed. For reference on symbols of configuration items see Section 5.13.2.2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_INIT* ioctl command is implemented as a function.

Example 5-365. DEC_INIT

```
ioctl( DEC_0, DEC_INIT, NULL );
```

This code initializes Quadrature Decoder 0 module with the initialization values from *appconfig.h*.

```
ioctl( DEC_1, DEC_INIT, NULL );
```

This code initializes Quadrature Decoder 1 module with the initialization values from *appconfig.h*.

5.13.3.2 DEC_INT_ENABLE - enable Quadrature Decoder interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INT_ENABLE, UWord16 param);
```

Arguments:

Table 5-409. DEC_INT_ENABLE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the Quadrature Decoder interrupt(s). Use consolidation of these predefined constants: DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT

Description: The *DEC_INT_ENABLE* ioctl command enables the Quadrature Decoder interrupt(s). There are three Quadrature Decoder interrupts: the Home Signal Interrupt, the Index Pulse Interrupt and the Watchdog Timeout Interrupt. This command sets the corresponding enable interrupt bits in the Decoder Control Register (DECCR). These bits are the Home Interrupt Enable (HIE), bit 14, when DEC_HOME is used as a parameter, the Index Pulse Interrupt Enable (XIE), bit 7, when DEC_INDEX is used as a parameter and the Watchdog Timeout Interrupt Enable (DIE), bit 3, when DEC_WDTIMEOUT is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-366. DEC_INT_ENABLE

```
ioctl( DEC_0, DEC_INT_ENABLE, DEC_INDEX );
```

This code enables the Quadrature Decoder 0 Index Pulse interrupt.

```
ioctl( DEC_1, DEC_INT_ENABLE, DEC_INDEX | DEC_WDTIMEOUT );
```

This code enables the Quadrature Decoder 1 Index Pulse and Watchdog Timeout interrupts.

5.13.3.3 DEC_INT_DISABLE - disable Quadrature Decoder interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-410. DEC_INT_DISABLE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the Quadrature Decoder interrupt(s). Use consolidation of these predefined constants: DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT

Description: The *DEC_INT_DISABLE* ioctl command disables the Quadrature Decoder interrupt(s). There are three Quadrature Decoder interrupts: the Home Signal Interrupt, the Index Pulse Interrupt and the Watchdog Timeout Interrupt. This command clears the corresponding enable interrupt bits in the Decoder Control Register (DECCR). These bits are the Home Interrupt Enable (HIE), bit 14, when DEC_HOME is used as a parameter, the Index Pulse Interrupt Enable (XIE), bit 7, when DEC_INDEX is used as a parameter and the Watchdog Timeout Interrupt Enable (DIE), bit 3, when DEC_WDTIMEOUT is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-367. DEC_INT_DISABLE

```
ioctl( DEC_0, DEC_INT_DISABLE, DEC_HOME );
```

This code disables the Quadrature Decoder 0 Home Signal interrupt.

```
ioctl( DEC_1, DEC_INT_DISABLE, DEC_INDEX | DEC_WDTIMEOUT );
```

This code disables the Quadrature Decoder 1 Index Pulse and Watchdog Timeout interrupts.

5.13.3.4 DEC_INT_REQUEST_CLEAR - clear Quadrature Decoder interrupt flag(s)

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INT_REQUEST_CLEAR,
                UWord16 param);
```

Arguments:

Table 5-411. DEC_INT_REQUEST_CLEAR ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the Quadrature Decoder interrupt flag(s). Use consolidation of these predefined constants: DEC_HOME | DEC_INDEX | DEC_WDTIMEOUT

Description: The *DEC_INT_REQUEST_CLEAR* ioctl command clears the selected Quadrature Decoder interrupt flag(s). There are three Quadrature Decoder interrupts: the Home Signal Interrupt, the Index Pulse Interrupt and the Watchdog Timeout Interrupt. This command clears the corresponding enable interrupt bits in the Decoder Control Register (DECCR). These bits are the HOME Signal Transition Interrupt Request (HIRQ), bit 15, when DEC_HOME is used as a parameter, the Index Pulse Interrupt Request (XIRQ), bit 8, when DEC_INDEX is used as a parameter and the Watchdog Timeout Interrupt Request (DIRQ), bit 4, when DEC_WDTIMEOUT is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_INT_REQUEST_CLEAR* ioctl command is implemented as a macro.

Example 5-368. DEC_INT_REQUEST_CLEAR

```
ioctl( DEC_0, DEC_INT_REQUEST_CLEAR, DEC_INDEX );
```

This code clears the Quadrature Decoder 0 Index Pulse interrupt request flag.

5.13.3.5 DEC_CLEAR_HOME_INT_REQUEST - clear HOME signal interrupt request flag

Call(s):

```
void ioctl(const int *pModuleBase, DEC_CLEAR_HOME_INT_REQUEST,
           NULL);
```

Arguments:

Table 5-412. DEC_CLEAR_HOME_INT_REQUEST ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_CLEAR_HOME_INT_REQUEST* ioctl command clears the HOME signal interrupt request flag. This command clears the HIRQ bit (Bit 15) in the Decoder Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_CLEAR_HOME_INT_REQUEST* ioctl command is implemented as a macro.

Example 5-369. DEC_CLEAR_HOME_INT_REQUEST

```
ioctl( DEC_0, DEC_CLEAR_HOME_INT_REQUEST, NULL );
```

This code clears the Quadrature Decoder 0 HOME signal interrupt request flag.

5.13.3.6 DEC_CLEAR_INDEX_PULSE_INT_REQUEST - clear INDEX pulse interrupt request flag

Call(s):

```
void ioctl(const int *pModuleBase,
           DEC_CLEAR_INDEX_PULSE_INT_REQUEST, NULL);
```

Arguments:

Table 5-413. DEC_CLEAR_INDEX_PULSE_INT_REQUEST ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_CLEAR_INDEX_PULSE_INT_REQUEST* ioctl command clears the INDEX pulse interrupt request flag. This command clears the XIRQ bit (Bit 8) in the Decoder Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_CLEAR_INDEX_PULSE_INT_REQUEST* ioctl command is implemented as a macro.

Example 5-370. DEC_CLEAR_INDEX_PULSE_INT_REQUEST

```
ioctl( DEC_0, DEC_CLEAR_INDEX_PULSE_INT_REQUEST, NULL );
```

This code clears the Quadrature Decoder 0 INDEX pulse interrupt request flag.

5.13.3.7 DEC_CLEAR_WATCHDOG_INT_REQUEST - clear watchdog time-out request flag

Call(s):

```
void ioctl(const int *pModuleBase,
           DEC_CLEAR_WATCHDOG_INT_REQUEST, NULL);
```

Arguments:

Table 5-414. DEC_CLEAR_WATCHDOG_INT_REQUEST ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_CLEAR_WATCHDOG_INT_REQUEST* ioctl command clears the watchdog time-out interrupt request flag. This command clears the DIRQ bit (Bit 4) in the Decoder Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_CLEAR_WATCHDOG_INT_REQUEST* ioctl command is implemented as a macro.

Example 5-371. DEC_CLEAR_WATCHDOG_INT_REQUEST

```
ioctl( DEC_0, DEC_CLEAR_WATCHDOG_INT_REQUEST, NULL );
```

This code clears the Quadrature Decoder 0 watchdog time-out interrupt request flag.

5.13.3.8 DEC_HOME_TRIGGERED_INIT - enable or disable the position counter to be initialized by the HOME signal

Call(s):

```
void ioctl(const int *pModuleBase, DEC_HOME_TRIGGERED_INIT,
           UWord16 param);
```

Arguments:

Table 5-415. DEC_HOME_TRIGGERED_INIT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable initialization by the HOME signal or DEC_DISABLE to disable it.

Description: The *DEC_HOME_TRIGGERED_INIT* ioctl command enables or disables initialization of the Upper and Lower Position Counter Registers (UPOS, LPOS) with the HOME signal. This command sets the HIP bit (Bit 13) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_HOME_TRIGGERED_INIT* ioctl command is implemented as a macro.

Example 5-372. DEC_HOME_TRIGGERED_INIT

```
ioctl( DEC_0, DEC_HOME_TRIGGERED_INIT, DEC_ENABLE );
```

This code enables initialization of the Quadrature Decoder 0 position counter by the HOME signal.

```
ioctl( DEC_1, DEC_HOME_TRIGGERED_INIT, DEC_DISABLE );
```

This code disables initialization of the Quadrature Decoder 1 position counter by the HOME signal.

5.13.3.9 DEC_HOME_EDGE - set the edge of the HOME signal to initialize the position counter

Call(s):

```
void ioctl(const int *pModuleBase, DEC_HOME_EDGE, UWord16 param);
```

Arguments:

Table 5-416. DEC_HOME_EDGE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired operation. Use DEC_POSITIVE to set the positive going edge to initialize the position counter or DEC_NEGATIVE to set the negative going edge to initialize the position counter.

Description: The *DEC_HOME_EDGE* ioctl command determines whether the positive or the negative going edge of the HOME signal triggers the initialization of the Upper and the Lower Position Counter Registers (UPOS, LPOS). This command sets the HNE bit (Bit 12) in the Decoder Control Register when DEC_NEGATIVE is used as a parameter. This command clears the above mentioned bit in case of DEC_POSITIVE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_HOME_EDGE* ioctl command is implemented as a macro.

Example 5-373. DEC_HOME_EDGE

```
ioctl( DEC_0, DEC_HOME_EDGE, DEC_POSITIVE );
```

This code sets the positive going edge of the HOME signal to initialize the Quadrature Decoder 0 position counters.

```
ioctl( DEC_1, DEC_HOME_EDGE, DEC_NEGATIVE );
```

This code sets the negative going edge of the HOME signal to initialize the Quadrature Decoder 1 position counters.

5.13.3.10 DEC_SOFTWARE_TRIGGERED_INIT - initialize the position counter

Call(s):

```
void ioctl(const int *pModuleBase, DEC_SOFTWARE_TRIGGERED_INIT,
           NULL);
```

Arguments:

Table 5-417. DEC_SOFTWARE_TRIGGERED_INIT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_SOFTWARE_TRIGGERED_INIT* ioctl command initializes the Upper and the Lower Position Counter Registers (UPOS, LPOS) by the values stored in the Upper and the Lower Initialization Registers. This command sets the SWIP bit (Bit 11) in the Decoder Control Register.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_SOFTWARE_TRIGGERED_INIT* ioctl command is implemented as a macro.

Example 5-374. DEC_SOFTWARE_TRIGGERED_INIT

```
ioctl( DEC_0, DEC_SOFTWARE_TRIGGERED_INIT, NULL );
```

This code initializes the Quadrature Decoder 0 position counters.

5.13.3.11 DEC_DIRECTION_COUNTING_ENABLE - change direction of counting

Call(s):

```
void ioctl(const int *pModuleBase, DEC_DIRECTION_COUNTING_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-418. DEC_DIRECTION_COUNTING_ENABLE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired operation. Use DEC_REVERSE to reverse the interpretation of the quadrature signal or DEC_NORMAL to count normally.

Description: The *DEC_DIRECTION_COUNTING_ENABLE* ioctl command reverses the interpretation of the quadrature signal, i.e. it changes the direction of count. This command sets the REV bit (Bit 10) in the Decoder Control Register when DEC_REVERSE is used as a parameter. This command clears the above mentioned bit in case of DEC_NORMAL.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_DIRECTION_COUNTING_ENABLE* ioctl command is implemented as a macro.

Example 5-375. DEC_DIRECTION_COUNTING_ENABLE

```
ioctl( DEC_0, DEC_DIRECTION_COUNTING_ENABLE, DEC_REVERSE );
```

This code reverses the direction of counting of quadrature signals for the Quadrature Decoder 0 module.

5.13.3.12 DEC_SINGLE_PHASE_COUNT - bypass the Quadrature Decoder logic

Call(s):

```
void ioctl(const int *pModuleBase, DEC_SINGLE_PHASE_COUNT,
           UWord16 param);
```

Arguments:

Table 5-419. DEC_SINGLE_PHASE_COUNT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired operation. Use DEC_ENABLE to bypass the Quadrature Decoder logic or DEC_DISABLE to use standard Quadrature Decoder logic.

Description: The *DEC_SINGLE_PHASE_COUNT* ioctl command enables or disables the Quadrature Decoder logic.When the Quadrature Decoder logic is bypassed, the PHASEA signal is used as a single phase pulse stream and PHASEB is ignored. This command sets the PH1 bit (Bit 9) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_SINGLE_PHASE_COUNT* ioctl command is implemented as a macro.

Example 5-376. DEC_SINGLE_PHASE_COUNT

```
ioctl( DEC_0, DEC_SINGLE_PHASE_COUNT, DEC_ENABLE );
```

This code bypasses the Quadrature Decoder 0 logic.

```
ioctl( DEC_1, DEC_SINGLE_PHASE_COUNT, DEC_DISABLE );
```

This code sets the Quadrature Decoder 1 to use the standard logic.

5.13.3.13 DEC_INDEX_TRIGGERED_INIT - enable or disable the position counter to be initialized by the INDEX pulse

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INDEX_TRIGGERED_INIT,
           UWord16 param);
```

Arguments:

Table 5-420. DEC_INDEX_TRIGGERED_INIT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable initialization by the INDEX pulse or DEC_DISABLE to disable it.

Description: The *DEC_INDEX_TRIGGERED_INIT* ioctl command enables or disables initialization of the Upper and the Lower Position Counter Registers (UPOS, LPOS) using the INDEX pulse. This command sets the XIP bit (Bit 6) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_INDEX_TRIGGERED_INIT* ioctl command is implemented as a macro.

Example 5-377. DEC_INDEX_TRIGGERED_INIT

```
ioctl( DEC_0, DEC_INDEX_TRIGGERED_INIT, DEC_ENABLE );
```

This code enables initialization of Quadrature Decoder 0 position counter by the INDEX pulse.

```
ioctl( DEC_1, DEC_INDEX_TRIGGERED_INIT, DEC_DISABLE );
```

This code disables initialization of Quadrature Decoder 1 position counter by the INDEX pulse.

5.13.3.14 DEC_INDEX_EDGE - set the edge of the INDEX pulse to initialize the position counter

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INDEX_EDGE, UWord16 param);
```

Arguments:

Table 5-421. DEC_INDEX_EDGE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired operation. Use DEC_POSITIVE to set the positive going edge to initialize the position counter or DEC_NEGATIVE to set the negative going edge to initialize the position counter.

Description: The *DEC_INDEX_EDGE* ioctl command determines whether the positive or the negative going edge of the INDEX pulse triggers the initialization of the Upper and Lower Position Counter Registers (UPOS, LPOS). This command sets the XNE bit (Bit 5) in the Decoder Control Register when DEC_NEGATIVE is used as a parameter. This command clears the above mentioned bit in case of DEC_POSITIVE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_INDEX_EDGE* ioctl command is implemented as a macro.

Example 5-378. DEC_INDEX_EDGE

```
ioctl( DEC_0, DEC_INDEX_EDGE, DEC_POSITIVE );
```

This code causes the positive going edge of the HOME signal to initialize the Quadrature Decoder 0 position counters.

```
ioctl( DEC_1, DEC_INDEX_EDGE, DEC_NEGATIVE );
```

This code causes the negative going edge of the HOME signal to initialize the Quadrature Decoder 1 position counters.

5.13.3.15 DEC_WATCHDOG - enable or disable watchdog timer

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WATCHDOG, UWord16 param);
```

Arguments:

Table 5-422. DEC_WATCHDOG ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable the watchdog timer or DEC_DISABLE to disable it.

Description: The *DEC_WATCHDOG* ioctl command enables or disables the watchdog timer to monitor PHASEA and PHASEB inputs for motor movement. This command sets the WDE bit (Bit 2) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_WATCHDOG* ioctl command is implemented as a macro.

Example 5-379. DEC_WATCHDOG

```
ioctl( DEC_0, DEC_WATCHDOG, DEC_ENABLE );
```

This code enables the Quadrature Decoder 0 watchdog timer.

```
ioctl( DEC_1, DEC_WATCHDOG, DEC_DISABLE );
```

This code disables the Quadrature Decoder 1 watchdog timer.

5.13.3.16 DEC_SWITCH_MATRIX - select the Switch Matrix Mode

Call(s):

```
void ioctl(const int *pModuleBase, DEC_SWITCH_MATRIX,
           UWord16 param);
```

Arguments:

Table 5-423. DEC_SWITCH_MATRIX ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired switch matrix mode. Use one of the predefined constants: DEC_MODE_0 / DEC_MODE_1 / DEC_MODE_2

Description: The *DEC_SWITCH_MATRIX* ioctl command selects the Switch Matrix Mode connecting inputs to the Timer module, see Table 5-424. This command manipulates the MODE bits (Bit 1-0) in the Decoder Control Register.

Table 5-424. Switch Matrix Mode

param	Modes
DEC_MODE_0	Mode0: Input Captures connected to the four input pins
DEC_MODE_1	Mode1: Input Captures connected to the filtered versions of the four input pins
DEC_MODE_2	Mode2: PHASEA input connected to both channels 0 and 1 of the Timer to allow capture of both rising and falling edges; PHASEB input connected to both channels 2 and 3 of the Timer

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *DEC_SWITCH_MATRIX* ioctl command is implemented as a macro.

Example 5-380. DEC_SWITCH_MATRIX

```
ioctl(DEC_0, DEC_SWITCH_MATRIX, DEC_MODE_1);
```

This code sets Mode1 for the Quadrature Decoder 0 module operation.

5.13.3.17 DEC_WRITE_FILTER - set Filter Interval Register

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WRITE_FILTER,
           UWord16 param);
```

Arguments:

Table 5-425. DEC_WRITE_FILTER ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Value representing filter interval value.

Description: The *DEC_WRITE_FILTER* ioctl command writes the value (*param*) to the Filter Interval Register (FIR). This value represents the filter interval periods in number of IP Bus clock periods. See the **MC56F8300 Peripheral User Manual** for more details.

Returns: None.

Range Issues: *param* must be within <0x000, 0x00ff>.

Special Issues: None.

Design/Implementation: The *DEC_WRITE_FILTER* ioctl command is implemented as a macro.

Example 5-381. DEC_WRITE_FILTER

```
UWord16    filterValue;

filterValue = 0x00ff;

ioctl( DEC_0, DEC_WRITE_FILTER, filterValue );
```

This code writes new a value to the Quadrature Decoder 0 Filter Interval Register.

```
ioctl( DEC_1, DEC_WRITE_FILTER, 0x007f );
```

This code writes 0x007f to the Quadrature Decoder 1 Filter Interval Register.

5.13.3.18 DEC_WRITE_WATCHDOG_TIMEOUT - set Filter Interval Register

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WRITE_WATCHDOG_TIMEOUT,
           UWord16 param);
```

Arguments:

Table 5-426. DEC_WRITE_WATCHDOG_TIMEOUT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Value representing time-out count in the number of clock cycles.

Description: The *DEC_WRITE_WATCHDOG_TIMEOUT* ioctl command writes the value (*param*) to the Watchdog Time-out Register (WTR). This value represents the time-out count as the number of clock cycles plus one, that the watchdog timer will count before timing out. It optionally generates an interrupt. See the **MC56F8300 Peripheral User Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_WRITE_WATCHDOG_TIMEOUT* ioctl command is implemented as a macro.

Example 5-382. DEC_WRITE_WATCHDOG_TIMEOUT

```
UWord16    timeOutValue;

timeOutValue = 0x00ff;

ioctl( DEC_0, DEC_WRITE_WATCHDOG_TIMEOUT, timeOutValue );
```

This code writes a new value to the Quadrature Decoder 0 Watchdog Time-out Register.

```
ioctl( DEC_1, DEC_WRITE_WATCHDOG_TIMEOUT, 0x007f );
```

This code writes 0x007f to the Quadrature Decoder 1 Watchdog Time-out Register.

5.13.3.19 DEC_READ_POSITION_DIFFERENCE - return the content of the Position Difference Counter Register

Call(s):

```
Word16 ioctl(const int *pModuleBase,
             DEC_READ_POSITION_DIFFERENCE, NULL);
```

Arguments:

Table 5-427. DEC_READ_POSITION_DIFFERENCE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_READ_POSITION_DIFFERENCE* ioctl command returns the content of the Position Difference Counter Register (POSD).

Returns: content of the Position Difference Counter Register as Word16.

Range Issues: None.

Special Issues: Causes to copy all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_READ_POSITION_DIFFERENCE* ioctl command is implemented as a macro.

Example 5-383. DEC_READ_POSITION_DIFFERENCE

```
Word16     posDiff;

posDiff = ioctl( DEC_0, DEC_READ_POSITION_DIFFERENCE, NULL );
```

This code stores the content of the Quadrature Decoder 0 Position Difference Counter Register to variable posDiff.

```
posDiff = ioctl( DEC_1, DEC_READ_POSITION_DIFFERENCE, NULL );
```

This code stores the content of the Quadrature Decoder 1 Position Difference Counter Register to variable posDiff.

5.13.3.20 DEC_READ_REVOLUTION - return the content of the Revolution Counter Register

Call(s):

```
Word16 ioctl(const int *pModuleBase, DEC_READ_REVOLUTION, NULL);
```

Arguments:

Table 5-428. DEC_READ_REVOLUTION ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_READ_REVOLUTION* ioctl command returns the content of the Revolution Counter Register (REV).

Returns: content of the Revolution Counter Register as Word16.

Range Issues: None.

Special Issues: Causes to copy all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_READ_REVOLUTION* ioctl command is implemented as a macro.

Example 5-384. DEC_READ_REVOLUTION

```
Word16     revNum;

revNum = ioctl( DEC_0, DEC_READ_REVOLUTION, NULL );
```

This code stores the content of the Quadrature Decoder 0 Revolution Counter Register to variable revNum.

```
revNum = ioctl( DEC_1, DEC_READ_REVOLUTION, NULL );
```

This code stores the content of the Quadrature Decoder 1 Revolution Counter Register to variable revNum.

5.13.3.21 DEC_WRITE_REVOLUTION - set the Revolution Counter Register

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WRITE_REVOLUTION,
           Word16 param);
```

Arguments:

Table 5-429. DEC_WRITE_REVOLUTION ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Value representing the number of revolutions.

Description: The *DEC_WRITE_REVOLUTION* ioctl command writes the value (*param*) to the Revolution Counter Register (REV). This value represents the required number of revolutions.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_WRITE_REVOLUTION* ioctl command is implemented as a macro.

Example 5-385. DEC_WRITE_REVOLUTION

```
Word16     revNum;

revNum = 0x700f;

ioctl( DEC_0, DEC_WRITE_REVOLUTION, revNum );
```

This code writes a new value to the Quadrature Decoder 0 Revolution Counter Register.

```
ioctl( DEC_1, DEC_WRITE_REVOLUTION, 0x0fff );
```

This code writes 0x0fff to the Quadrature Decoder 1 Revolution Counter Register.

5.13.3.22 DEC_READ_POSITION - read the content of the Upper and Lower Position Counter Registers

Call(s):

```
void ioctl(const int *pModuleBase, DEC_READ_POSITION,
           decoder_uReg32bit *pParam);
```

Arguments:

Table 5-430. DEC_READ_POSITION ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
*pParam	out	Pointer to the union of the *decoder_uReg32bit* type representing the content of the position registers as a 32bit value or two 16bit values

Description: The *DEC_READ_POSITION* ioctl command reads the content of the Upper and the Lower Position Counter Register (UPOS, LPOS).

Returns: None.

Range Issues: None.

Special Issues: Causes to copy all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_READ_POSITION* ioctl command is implemented as a macro.

Example 5-386. DEC_READ_POSITION

```
decoder_uReg32bit positionReg32;

ioctl( DEC_0, DEC_READ_POSITION, &positionReg32 );
```

This code stores the content of the Quadrature Decoder 0 Position Counter Registers to variable positionReg32.

```
ioctl( DEC_1, DEC_READ_POSITION, &positionReg32 );
```

This code stores the content of the Quadrature Decoder 1 Position Counter Registers to variable positionReg32.

5.13.3.23 DEC_WRITE_POSITION - set the Upper and Lower Position Counter Registers

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WRITE_POSITION,
           Word32 param);
```

Arguments:

Table 5-431. DEC_WRITE_POSITION ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Value representing the number of encoder pulses.

Description: The *DEC_WRITE_POSITION* ioctl command writes the value (*param*) to the Lower and the Upper Position Counter Registers (UPOS, LPOS). This value represents the required number of encoder pulses. This command writes the value to the Upper and the Lower Initialization Register followed by initialization of the position registers with the software trigger command.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_WRITE_POSITION* ioctl command is implemented as a macro.

Example 5-387. DEC_WRITE_POSITION

```
Word32 position

position = 0x0abc1234;

ioctl( DEC_0, DEC_WRITE_POSITION, position );
```

This code writes a new value to the Quadrature Decoder 0 Position Counter Registers.

```
ioctl( DEC_1, DEC_WRITE_POSITION, 0x0123ffff );
```

This code writes 0x0123ffff to the Quadrature Decoder 1 Position Counter Registers.

5.13.3.24 DEC_WRITE_INIT_STATE - set the Upper and Lower Initialization Registers

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WRITE_INIT_STATE,
           Word32 param);
```

Arguments:

Table 5-432. DEC_WRITE_INIT_STATE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Value representing the number of encoder pulses.

Description: The *DEC_WRITE_INIT_STATE* ioctl command writes the value (*param*) to the Lower and the Upper Initialization Registers (UIR, LIR). This value represents the initialization number of encoder pulses.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_WRITE_INIT_STATE* ioctl command is implemented as a macro.

Example 5-388. DEC_WRITE_INIT_STATE

```
Word32 position

position = 0x0abc1234;

ioctl( DEC_0, DEC_WRITE_INIT_STATE, position );
```

This code writes a new value to the Quadrature Decoder 0 Initialization Registers.

```
ioctl( DEC_1, DEC_WRITE_INIT_STATE, 0x0123ffff );
```

This code writes 0x0123ffff to the Quadrature Decoder 1 Initialization Registers.

5.13.3.25 DEC_READ_MONITOR_REG - return the content of the Input Monitor Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DEC_READ_MONITOR_REG,
          NULL);
```

Arguments:

Table 5-433. DEC_READ_MONITOR_REG ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_READ_MONITOR_REG* ioctl command returns the content of the Input Monitor Register (IMR).

Returns: content of the Input Monitor Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_READ_MONITOR_REG* ioctl command is implemented as a macro.

Example 5-389. DEC_READ_MONITOR_REG

```
UWord16    monitor;

monitor = ioctl( DEC_0, DEC_READ_MONITOR_REG, NULL );
```

This code stores the content of the Quadrature Decoder 0 Input Monitor Register to variable monitor.

5.13.3.26 DEC_GET_RAW_ENCSIGNALS - return the raw version of INDEX, PHASEB and PHASEA encoder signals

Call(s):

```
decoder_uEncSignals.Value ioctl(const int *pModuleBase,
           DEC_GET_RAW_ENCSIGNALS,
           NULL);
```

Arguments:

Table 5-434. DEC_GET_RAW_ENCSIGNALS ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_GET_RAW_ENCSIGNALS* ioctl command returns the raw version of INDEX (INDEX - Bit 1), PHASEB (PHB - Bit2) and PHASEA (PHA - Bit3) encoder signals from the Input Monitor Register (IMR).

Returns: the actual state of Encoder device signals INDEX, PHASEB and PHASEA as UWord16 or as decoder_sEncSignals structure members.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_GET_RAW_ENCSIGNALS* ioctl command is implemented as a macro.

Example 5-390. DEC_GET_RAW_ENCSIGNALS

```
decoder_uEncSignals   encSignals;

encSignals.Value = ioctl( DEC_0, DEC_GET_RAW_ENCSIGNALS, NULL );
```

This code stores the raw version of the INDEX, PHASEB and PHASEA encoder signals from the Quadrature Decoder 0 Input Monitor Register to variable encSignals.

5.13.3.27 DEC_GET_FILTERED_ENCSIGNALS - return the filtered version of INDEX, PHASEB and PHASEA encoder signals

Call(s):

```
decoder_uEncSignals.Value ioctl(const int *pModuleBase,
           DEC_GET_FILTERED_ENCSIGNALS,
           NULL);
```

Arguments:

Table 5-435. DEC_GET_FILTERED_ENCSIGNALS ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_GET_FILTERED_ENCSIGNALS* ioctl command returns the filtered version of INDEX (INDEX - Bit 1), PHASEB (PHB - Bit2) and PHASEA (PHA - Bit3) encoder signals from the Input Monitor Register (IMR).

Returns: the actual state of Encoder device signals INDEX, PHASEB and PHASEA as UWord16 or as decoder_sEncSignals structure members.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_GET_FILTERED_ENCSIGNALS* ioctl command is implemented as a macro.

Example 5-391. DEC_GET_FILTERED_ENCSIGNALS

```
decoder_uEncSignals   encSignals;

encSignals.Value = ioctl( DEC_0, DEC_GET_FILTERED_ENCSIGNALS,\
NULL );
```

This code stores the filtered version of the INDEX, PHASEB and PHASEA encoder signals from the Quadrature Decoder 0 Input Monitor Register to variable encSignals.

5.13.3.28 DEC_READ_HOLD_DATA_REGS - hold and store the content of the Upper and the Lower Position Counter Registers, the Position Difference Counter Register and the Revolution Counter Register

Call(s):

```
void ioctl(const int *pModuleBase, DEC_READ_HOLD_DATA_REGS,
           decoder_sState *pParam);
```

Arguments:

Table 5-436. DEC_READ_HOLD_DATA_REGS ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
*pParam	out	Pointer to the union of the *decoder_sState* type representing the content of the position registers, position difference register and revolution register.

Description: The *DEC_READ_HOLD_DATA_REGS* ioctl command holds and stores the content of the Upper and the Lower Position Counter Register (UPOS, LPOS), the Position Difference Counter Register (POSD) and the Revolution Counter Register (REV) to the corresponding structure members. This command provides a snapshot of the counters' values to maintain a consistent view of the Decoder data registers at the certain point in time.

Returns: None.

Range Issues: None.

Special Issues: Causes to copy all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_READ_HOLD_DATA_REGS* ioctl command is implemented as a macro.

Example 5-392. DEC_READ_HOLD_DATA_REGS

```
decoder_sState holdDataRegs;

ioctl( DEC_0, DEC_READ_HOLD_DATA_REGS, &holdDataRegs );
```

This code holds and stores the content of the Quadrature Decoder 0 Position Counter Registers, the Position Difference Register and the Revolution Counter Register to the holdDataRegs structure members.

```
ioctl( DEC_1, DEC_READ_HOLD_DATA_REGS, &holdDataRegs );
```

This code holds and stores the content of the Quadrature Decoder 1 Position Counter Registers, the Position Difference Register and the Revolution Counter Register to the holdDataRegs structure members.

5.13.3.29 DEC_READ_CONTROL_REG - return the content of the Input Monitor Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DEC_READ_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-437. DEC_READ_CONTROL_REG ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.

Description: The *DEC_READ_CONTROL_REG* ioctl command returns the content of the Decoder Control Register (DECCR).

Returns: the content of the Decoder Control Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_READ_MONITOR_REG* ioctl command is implemented as a macro.

Example 5-393. DEC_READ_MONITOR_REG

```
UWord16    cntrl;

cntrl = ioctl( DEC_0, DEC_READ_MONITOR_REG, NULL );
```

This code stores the content of the Quadrature Decoder 0 Control Register to the variable cntrl.

5.13.3.30 DEC_CALCULATE_SCALE_COEF - precalculate the scaling coefficients

Call(s):

```
void ioctl(const int *pModuleBase, DEC_CALCULATE_SCALE_COEF,
           decoder_sEncScale *pParam);
```

Arguments:

Table 5-438. DEC_CALCULATE_SCALE_COEF ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
*pParam	inout	Pointer to the structure of the *decoder_sEncScale* type containing the number of Encoder pulses and required scaling

Description: The *DEC_CALCULATE_SCALE_COEF* ioctl calculates the scaling coefficients needed for correct functionality of the DEC_GET_SCALED_POSITION and the DEC_GET_SCALED_POSITION_DIFFERENCE commands (i.e. this command must be executed before execution of above mentioned commands). The *decoder_sEncScale* members *EncPulses* and *RevolutionScale* must be filled up prior to calling this command. *EncPulses* represents the nominal number of Encoder pulses per revolution and *RevolutionScale* represents the number of revolutions to be reflected by the 16 bit register full range:

```
RevolutionScale = 0 represents a range +/- PI
RevolutionScale = 1 represents a range +/- 2PI
RevolutionScale = 2 represents a range +/- 4PI
etc.
```

Returns: None.

Range Issues: The maximum RevolutionScale value is determined by the following expression:

```
32767 > ((EncPulses * 4) * RevolutionScale)
```

When RevolutionScale is equal to 0, then the following expression must be valid:

```
32767 > (EncPulses * 2)
```

Special Issues: The *RevolutionScale* structure member has a meaning only for the DEC_GET_SCALED_POSITION_DIFFERENCE command.

Design/Implementation: The *DEC_CALCULATE_SCALE_COEF* ioctl command is implemented as a function.

Example 5-394. DEC_CALCULATE_SCALE_COEF

```
decoder_sEncScale decEncScale;
decEncScale.EncPulses = 1024; /* Encoder pulses per revolution */
decEncScale.RevolutionScale = 1; /* range +/-2PI to be represented
                                    by 16bit value */

ioctl( DEC_0, DEC_CALCULATE_SCALE_COEF, &decEncScale );
```

This code precalculates the scaling coefficients, based on specified parameters, for the subsequent usage.

5.13.3.31 DEC_GET_SCALED_POSITION - calculate an absolute position

Call(s):

```
Word32 ioctl(const int *pModuleBase, DEC_GET_SCALED_POSITION,
             decoder_sEncScale *pParam);
```

Arguments:

Table 5-439. DEC_GET_SCALED_POSITION ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
*pParam	in	Pointer to the structure of the *decoder_sEncScale* type contain the precalculated scaling coefficients.

Description: The *DEC_GET_SCALED_POSITION* ioctl calculates and returns an absolute position. The DEC_CALCULATE_SCALE_COEF command must be executed prior to this command. This command reads the content of the Revolution Counter Register (REV) and of the Upper and the Lower Position Counter Register (UPOS, LPOS).

Returns: a 32bit value, where the MSB part represents the number of revolutions (equals to the content of the Revolution Register) while the LSB part represents the portion of the current revolution, scaled into the range of a 16bit unsigned data.

Range Issues: None.

Special Issues: The correct functionality requires to fill the Initialization Register by value 0x00000000 and to enable initialization of the Position Counter by the INDEX signal (DEC_WRITE_INIT_STATE and DEC_INDEX_TRIGGERED_INIT commands) during the Quadrature Decoder initialization-configuration phase.

This command performs the copy of all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_GET_SCALED_POSITION* ioctl command is implemented as a function.

Example 5-395. DEC_GET_SCALED_POSITION

```
Word32     absPos;
decoder_sEncScale decEncScale;
ioctl( DEC_0, DEC_WRITE_INIT_STATE, 0x00000000 );
ioctl( DEC_0, DEC_INDEX_TRIGGERED_INIT, DEC_ENABLE );
/* see Example 5-394; suppose 1024 Encoder pulses per revolution
*/
decEncScale.EncPulses = 1024;
ioctl( DEC_0, DEC_CALCULATE_SCALE_COEF, &decEncScale );
/* suppose: Revolution Reg=0x0125 and Position Regs=0x00000800 */
absPos = ioctl(DEC_0,DEC_GET_SCALED_POSITION,&decEncScale);
/* absPos=0x0125.7fff, i.e. result represents 125.5 revolutions */
```

This code illustrates the usage of the DEC_GET_SCALED_POSITION command for the Quadrature Decoder 0 by one example.

5.13.3.32 DEC_GET_SCALED_POSITION_DIFFERENCE - calculate an relative position

Call(s):

```
Word16 ioctl(const int *pModuleBase,
DEC_GET_SCALED_POSITION_DIFFERENCE, decoder_sEncScale *pParam);
```

Arguments:

Table 5-440. DEC_GET_SCALED_POSITION_DIFFERENCE ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
*pParam	in	Pointer to the structure of the *decoder_sEncScale* type containing the precalculated scaling coefficients.

Description: The *DEC_GET_SCALED_POSITION_DIFFERENCE* ioctl command calculates and returns a relative position. The DEC_CALCULATE_SCALE_COEF command must be executed prior to this command. This command reads the content of the Position Difference Counter Register (POSD).

Returns: the scaled relative position - difference as Word16.

Range Issues: The returned 16bit signed value represents a range specified by the *RevolutionScale* structure member.

Special Issues: This command recalculates (scales) the content of the Position Difference Counter Register, which is automatically cleared when the Position Register is read.

This command perform the copy of all Quadrature Decoder counter registers' contents to their corresponding hold registers.

Design/Implementation: The *DEC_GET_SCALED_POSITION_DIFFERENCE* ioctl command is implemented as a macro.

Example 5-396. DEC_GET_SCALED_POSITION_DIFFERENCE

```
Word16     relPos;
decoder_sEncScale decEncScale;
/* see Example 5-394; suppose 1024 Encoder pulses per revolution
and revolutionScale equal to 1, i.e. range +/-2PI */
decEncScale.EncPulses = 1024;
decEncScale.RevolutionScale = 1;
ioctl( DEC_0, DEC_CALCULATE_SCALE_COEF, &decEncScale );
/* suppose: Position Difference Reg=2048 */
relPos = ioctl(DEC_0,DEC_GET_SCALED_POSITION_DIFFERENCE, \
                &decEncScale);
/* relPos=0x3fff, i.e. result represents 0.5 revolutions */
```

This code illustrates the usage of the DEC_GET_SCALED_POSITION_DIFFERENCE command for the Quadrature Decoder 0 by one example.

5.13.3.33 DEC_HOME_INT - enable or disable HOME signal interrupt

Call(s):

```
void ioctl(const int *pModuleBase, DEC_HOME_INT, UWord16 param);
```

Arguments:

Table 5-441. DEC_HOME_INT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable the HOME signal interrupt or DEC_DISABLE to disable it.

Description: The *DEC_HOME_INT* ioctl command enables or disables the HOME signal interrupt request. This command sets the HIE bit (Bit 14) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_HOME_INT* ioctl command is implemented as a macro.

Example 5-397. DEC_HOME_INT

```
ioctl( DEC_0, DEC_HOME_INT, DEC_ENABLE );
```

This code enables the Quadrature Decoder 0 HOME signal interrupt.

5.13.3.34 DEC_INDEX_PULSE_INT - enable or disable INDEX pulse interrupt

Call(s):

```
void ioctl(const int *pModuleBase, DEC_INDEX_PULSE_INT,
           UWord16 param);
```

Arguments:

Table 5-442. DEC_INDEX_PULSE_INT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable the INDEX pulse interrupt or DEC_DISABLE to disable it.

Description: The *DEC_INDEX_PULSE_INT* ioctl command enables or disables the INDEX pulse interrupt request. This command sets the XIE bit (Bit 7) in the Decoder Control Register when DEC_ENABLE is used as parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_INDEX_PULSE_INT* ioctl command is implemented as a macro.

Example 5-398. DEC_INDEX_PULSE_INT

```
ioctl( DEC_0, DEC_INDEX_PULSE_INT, DEC_ENABLE );
```

This code enables the Quadrature Decoder 0 INDEX pulse interrupt.

```
ioctl( DEC_1, DEC_INDEX_PULSE_INT, DEC_DISABLE );
```

This code disables the Quadrature Decoder 1 INDEX pulse interrupt.

5.13.3.35 DEC_WATCHDOG_INT - enable or disable watchdog time-out interrupt

Call(s):

```
void ioctl(const int *pModuleBase, DEC_WATCHDOG_INT,
           UWord16 param);
```

Arguments:

Table 5-443. DEC_WATCHDOG_INT ioctl call arguments

*pModuleBase	in	The Quadrature Decoder module identifier. Use DEC_0 or DEC_1. Note that DEC_1 is not available on certain chips.
param	in	Parameter to select the desired action. Use DEC_ENABLE to enable the watchdog time-out interrupt or DEC_DISABLE to disable it.

Description: The *DEC_WATCHDOG_INT* ioctl command enables or disables the watchdog time-out interrupt request. This command sets the DIE bit (Bit 3) in the Decoder Control Register when DEC_ENABLE is used as a parameter. This command clears the above mentioned bit in case of DEC_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *DEC_WATCHDOG_INT* ioctl command is implemented as a macro.

Example 5-399. DEC_WATCHDOG_INT

```
ioctl( DEC_0, DEC_WATCHDOG_INT, DEC_ENABLE );
```

This code enables the Quadrature Decoder 0 watchdog time-out interrupt.

```
ioctl( DEC_1, DEC_WATCHDOG_INT, DEC_DISABLE );
```

This code disables the Quadrature Decoder 1 watchdog time-out interrupt.

5.13.4 Quadrature Decoder Driver Application

The Quadrature Decoder driver application is designed for Motorola/Freescale EVM's and intended to illustrate the usage of this driver by a real example and also to verify the functionality. The general purpose LED associated with the GPIO pin indicates the Quadrature Decoder INDEX pulse interrupt and changes its state inside the interrupt service routine, i.e. when a new revolution is completed. Note: the user can spin the motor shaft manually. The Quadrature Decoder driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8346EVM\decoder_demo and consists of the application *decoder_demo.mcp* and the source code for the application *main.c*. See Table 5-444 for the LED name, Encoder connector and Encoder jumper setting specific to the individual EVM's.

Table 5-444. EVMs configuration

EVM	LED to indicate INDEX Pulse interrupt	Encoder connector	Jumper / position
MC56F8013DEMO	N/A	N/A	N/A
MC56F8013CB	N/A	N/A	N/A
MC56F8323EVM with Legacy Motor Interface Daughter Card (LMIDC)	green LED	Primary	JG4/2-3, 5-6, 8-9
56F8300DEMO	N/A	N/A	N/A
MC56F8346EVM with LMIDC	red LED	Secondary	JG5/ 2-3, 5-6, 8-9
MC56F8346CB with LMIDC	yellow LED	Secondary	JG5/ 2-3, 5-6, 8-9
MC56F8357EVM with LMIDC	red LED	Secondary	JG5/ 2-3, 5-6, 8-9
MC56F8367EVM with LMIDC	red LED	Secondary	JG5/ 2-3, 5-6, 8-9

Example 5-400. Quadrature Decoder Driver Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*   File generated by Graphical Configuration Tool
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1

/*.
    OCCS Configuration
----------------------------------------
    Core frequency: 60.000 MHz
    VCO frequency: 240.000 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.389 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
----------------------------------------
    SIM: Power Saving Modes: Stop enable , Wait enable
         Ext. bus driven when inactive: Disable
         OnCE clock to HawkV2 core: Enabled when core TAP enabled
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
```

```
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    Clock Output: CLKO select: sys_clk (OCCS) , CLKOUT mode: Tristated
    SEMI - CS0: Base address: 0x0, Blocksize: 128K , BYTE_EN: Both bytes enable , R/W:
Read / Write
                PS/DS select: PS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS1: Base address: 0x0, Blocksize: 128K , BYTE_EN: Lower byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS2: Base address: 0x0, Blocksize: 128K , BYTE_EN: Upper byte enable , R/W:
Read / Write
                PS/DS select: DS only , Write Wait States: 3, Read Wait States: 3
    SEMI - CS3: Base address: 0x0, Blocksize: 32K , BYTE_EN: Disable , R/W: Disable
                PS/DS select: Disable , Write Wait States: 23, Read Wait States: 23
.*/
#define SEMI_CSBAR0_INIT                  0x0005
#define SEMI_CSBAR1_INIT                  0x0005
#define SEMI_CSBAR2_INIT                  0x0005
#define SEMI_CSOR0_INIT                   0x1FC3
#define SEMI_CSOR1_INIT                   0x1BA3
#define SEMI_CSOR2_INIT                   0x1DA3
#define SIM_GPS_INIT                      0x0000

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                    0x0000
#define INT_VECTOR_ADDR_50                Index_ISR
#define INT_PRIORITY_LEVEL_50             INTC_LEVEL1

/*.
    DEC_0 Configuration
-----------------------------------------
    Mode (Sources of timer 0,1,2,3): Mode 0: PhA,PhB,Index,Home
    Direction of counting: Normal
    Home to initialize Position Counter: No , By positive edge
    Index Pulse to Initialize Position Counter: No , By positive edge
    Watchdog Operation: Disabled
    Bypass Quadrature Decoder Logic: No
    Watchdog timeout: 16.667 ms
    Filter interval period: 66.667 ms
    Interrupts: Home pulse: Disabled
                Index pulse: Disabled
                Watchdog timeout: Disabled
.*/
#define DEC_0_DECCR_INIT                  0x0000
#define DEC_0_FIR_INIT                    0x0003


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-401. Quadrature Decoder Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME:   main.c
*
* DESCRIPTION: Sample application demonstrating the use of Quadrature Decoder driver
*
*                          Each new revolution is indicated by toggling the state
*                          of the red LED; spin the motor shaft manually
*
*            NOTE:         The Secondary encoder connector (J4) on LMIDC must be used
*              to connect the motor encoder; jumper JG5 settings: 2-3, 5-6, 8-9
*
* TARGET:      MC56F8346 device
*
*****************************************************************************/

#include "qs.h"

#include "occs.h"
#include "sys.h"
#include "intc.h"
#include "gpio.h"
#include "cop.h"
#include "sci.h"
#include "spi.h"
#include "adc.h"
#include "fcan.h"
#include "qtimer.h"
#include "decoder.h"
#include "mc.h"
#include "pwm.h"
#include "tsensor.h"
#include "pcmaster.h"

// global variables
static  decoder_uReg32bit  PositionReg32;
static  UWord16            Revolutions = 0;

void Index_ISR(void);

/*****************************************************************************
             user defined ISR which is tied to the INDEX pulse, i.e. it is entered
once per revolution
*****************************************************************************/

#pragma interrupt on
void Index_ISR(void)
{
     /* read Revolution Register */
     Revolutions = ioctl( DEC_0, DEC_READ_REVOLUTION, NULL );

     /* toggle the red LED */
     ioctl(GPIO_C, GPIO_TOGGLE_PIN, BIT_3);

     /* clear INDEX pulse interrupt request flag */
     ioctl( DEC_0, DEC_CLEAR_INDEX_PULSE_INT_REQUEST, NULL );
}
```

```
#pragma interrupt off


/*****************************************************************************
            main
*****************************************************************************/

int main(void)
{
    /* initialize Decoder0 based on config items from appconfig.h */
    ioctl( DEC_0, DEC_INIT, NULL);

             /* configure GPIO C for LEDs */
    ioctl(GPIO_C, GPIO_SETAS_GPIO, BIT_0 | BIT_1 | BIT_2 | BIT_3 );
    ioctl(GPIO_C, GPIO_SETAS_OUTPUT, BIT_0 | BIT_1 | BIT_2 | BIT_3 );

    Revolutions = 0;
    /* write 0 to Revolution Counter Register */
    ioctl( DEC_0, DEC_WRITE_REVOLUTION, 0 );

    /* write INIT state value, enable initialization by INDEX */
    ioctl( DEC_0, DEC_WRITE_INIT_STATE, 0x00000000 );
    ioctl( DEC_0, DEC_INDEX_TRIGGERED_INIT, DEC_ENABLE );

    /* enable INDEX pulse interrupt */
    ioctl( DEC_0, DEC_INT_ENABLE, DEC_INDEX );

             /* configure Interrupt Controller (IPR) */
             ioctl(INTC, INTC_INIT, NULL);

             /* enable maskable interrupts in Status Register (SR), bits I1 and I0 */
             archEnableInt();

    while(1)
    {
        /* read periodically Position Register (L,U) */
        ioctl( DEC_0, DEC_READ_POSITION, &PositionReg32 );
             }
             return;
}
```

5.14 PWM Driver

This section describes the API for the MC56F83xx and MC56F80xx Pulse Width Modulator (PWM) on-chip module. The functionality of the PWM module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**..

5.14.1 Introduction

The MC56F83xx and MC56F80xx devices have up to two PWM modules (PWMA and PWMB), each with 6 PWM outputs, 3 Current Sense inputs and 4 Fault inputs, fault tolerant design with Deadtime insertion, supports both center and edge aligned modes. The PWM module can be configured for 1, 2 or 3 complementary pairs, the remaining outputs can then be used as independent channels.

This section describes the PWM driver software providing the lowest level software layer, interfacing the hardware with the software.

5.14.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.14.3.

Table 5-445. PWM Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F832x	MC56F83xx
PWM (PWM_BASE)	0xF020	0xF040	0xF0C0	N/A	N/A
PWM A (PWMA_BASE)	N/A	N/A	N/A	0xF140	0xF140
PWM B (PWMB_BASE)	N/A	N/A	N/A	N/A	0xF160

5.14.2.1 PWM Module Setting

The following expressions are useful for PWM module setting.

5.14.2.1.1 PWM Clock Calculation

```
PWM Clock Period = PWM Prescaler / IP Bus Clock Frequency
```

5.14.2.1.2 PWM Period and Frequency Calculation

for CENTER ALIGNMENT

```
PWM Period = PWM Modulus * PWM Clock Period * 2
```

for EDGE ALIGNMENT

```
PWM Period = PWM Modulus * PWM Clock Period
PWM Frequency = 1 / PWM Period
```

5.14.2.1.3 Deadtime Calculation

```
deadtime = (PWM Deadtime Reg. * PWM Clock Period) - 1 IP Bus Clock
(the one IP Bus Clock is not subtracted when PWM Prescaler is 1)
```

5.14.2.2 API Definition

The following header files are needed in order to use the PWM device driver:

Required Header File(s):

```
#include "qs.h"
#include "pwm.h"
```

The following information may be found in the header file *pwm.h*.

Public Data Structure(s):

```
typedef struct
{
     Word16  pwmChannel_0_Value;
     Word16  pwmChannel_2_Value;
     Word16  pwmChannel_4_Value;
}pwm_sComplementaryValues;

typedef struct
{
     Word16  pwmChannel_0_Value;
     Word16  pwmChannel_1_Value;
     Word16  pwmChannel_2_Value;
     Word16  pwmChannel_3_Value;
     Word16  pwmChannel_4_Value;
     Word16  pwmChannel_5_Value;
}pwm_sIndependentValues;

typedef struct
{
     pwm_tPWMSignalMask   SoftwareControlled;
     pwm_tPWMSignalMask   OutputControl;
}pwm_sOutputControl;

typedef struct
{
     Word16    DutyCycle;
     UWord16   Vlmode;
}pwm_sUpdateValueSetVlmode;

typedef UWord16 pwm_tPWMChannelSwap;


typedef struct
{
     pwm_tPWMSignalMask      Mask;
     pwm_tPWMChannelSwap    Swap;
}pwm_sChannelControl;

typedef UWord16 pwm_tPWMSignalMask;
```

Members:

Table 5-446. pwm_sComplementaryValues Data Structure Members

Member	Type	Description
pwmChannel_0_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_2_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_4_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.

Table 5-447. pwm_sIndependentValues Data Structure Members

Member	Type	Description
pwmChannel_0_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_1_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_2_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_3_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_4_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.
pwmChannel_5_Value	Word16	A data value representing a duty cycle or the desired value of the corresponding PWM Value register.

Table 5-448. pwm_sOutputControl Data Structure Members

Member	Type	Description
SoftwareControlled	pwm_tPWMSignalMask	A mask to enable software control of the corresponding PWM pins.
OutputControl	pwm_tPWMSignalMask	A mask to control the PWM output pins.

Table 5-449. pwm_sUpdateValueSetVlmode Data Structure Members

Member	Type	Description
DutyCycle	Word16	A data value representing a duty cycle corresponding to the PWM Value 0 register.
Vlmode	UWord16	A value representing the Value Register Load Mode.

Table 5-450. pwm_sChannelControl Data Structure Members

Member	Type	Description
Mask	pwm_tPWMSignalMask	A mask of the PWM logical channels.
Swap	pwm_tPWMChannelSwap	A mask to determine channel swapping operation.

5.14.2.3 Configuration Items

This section summarizes the symbols used in macro definitions for the static configuration of the PWM module by the driver initialization routine. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-472 for more details.

Table 5-451. PWM Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
PWM_x_PMCTL_INIT	UWord16	Initial value of the PWM Control Register.
PWM_x_PMFCTL_INIT	UWord16	Initial value of the PWM Fault Control Register.
PWM_x_PMOUT_INIT	UWord16	Initial value of the PWM Output Control Register.
PWM_x_PWMCM_INIT	UWord16	Initial value of the PWM Counter Modulo Register.
PWM_x_PWMVAL0_INIT	UWord16	Initial value of the PWM Value Register 0.

Table 5-451. PWM Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
PWM_x_PWMVAL1_INIT	UWord16	Initial value of the PWM Value Register 1.
PWM_x_PWMVAL2_INIT	UWord16	Initial value of the PWM Value Register 2.
PWM_x_PWMVAL3_INIT	UWord16	Initial value of the PWM Value Register 3.
PWM_x_PWMVAL4_INIT	UWord16	Initial value of the PWM Value Register 4.
PWM_x_PWMVAL5_INIT	UWord16	Initial value of the PWM Value Register 5.
only on MC56F83xx: PWM_x_PMDEADTM_INIT	UWord16	Initial value of the PWM Deadtime Register.
only on MC56F80xx: PWM_x_PMDEADTM0_INIT	UWord16	Initial value of the PWM Deadtime Register 0.
only on MC56F80xx: PWM_x_PMDEADTM1_INIT	UWord16	Initial value of the PWM Deadtime Register 1.
PWM_x_PMDISMAP1_INIT	UWord16	Initial value of the PWM Disable Mapping Register 1.
PWM_x_PMDISMAP2_INIT	UWord16	Initial value of the PWM Disable Mapping Register 2.
PWM_x_PMCFG_INIT	UWord16	Initial value of the PWM Configure Register.
PWM_x_PMCCR_INIT	UWord16	Initial value of the PWM Channel Control Register.
PWM_x_PMICCR_INIT	UWord16	Initial value of the PWM Internal Correction Control Register.
only on MC56F80xx: PWM_x_PMSRC_INIT	UWord16	Initial value of the PWM Source Control Register.
only on MC56F802x/3x and MC56F800x: PWM_x_SYNC_INIT	UWord16	Contents of the PWM Synchronization Window Register.
only on MC56F802x/3x and MC56F800x: PWM_x_FFILT0_INIT PWM_x_FFILT1_INIT PWM_x_FFILT2_INIT PWM_x_FFILT3_INIT	UWord16	Contents of the PWM Filter Registers 0-3.

Note - PWM_*x* should be replaced by PWM_A for the PWM A module and PWM_B for the PWM B module or by the PWM on MC56F80xx.

5.14.2.4 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
```

Description: The *ioctl* call "changes" PWM device modes or accesses the PWM register(s).

Arguments:

Table 5-452. PWM Driver Arguments - ioctl

pModuleBase	in	PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx.
cmd	in	Commands found in *pwm.h* which are used to modify the PWM module status and control registers. See Table 5-453.
pParam, param	in, inout	Used to pass the relevant data to ioctl function call.

Items Separator Convention:

```
/  only one of the specified items is allowed
|  combination of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-453. ioctl commands

cmd	param	Return	Description
PWM_INIT	NULL	None	Initializes PWM module by data from configuration file (appconfig.h).

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_SET_RELOAD_FREQUENCY	PWM_RELOAD_OPPORTUNITY_*x where x is in the range 1-16*	None	Sets the PWM Reload Frequency by selecting certain order of opportunity.
PWM_HALF_CYCLE_RELOAD	PWM_ENABLE / PWM_DISABLE	None	Half Cycle Reload enable or disable.
PWM_SET_CURRENT_POLARITY	(PWM_IPOL0 | PWM_IPOL1 | PWM_IPOL2) / PWM_ZERO_MASK	None	Sets the selected Current Polarity Bits.
PWM_SET_PRESCALER	PWM_PRESCALER_DIV_1 / PWM_PRESCALER_DIV_2 / PWM_PRESCALER_DIV_4 / PWM_PRESCALER_DIV_8	None	Sets the PWM clock frequency as an fraction of IP Bus frequency.
PWM_RELOAD_INT	PWM_ENABLE / PWM_DISABLE	None	PWM Reload Interrupt enable or disable.
PWM_SET_CURRENT_SENSING	PWM_CORRECTION_NO / PWM_CORRECTION_SOFTWARE / PWM_CORRECTION_DURING_ DEADTIME / PWM_CORRECTION_DURING_ CYCLE	None	Selects the top/bottom correction scheme.
PWM_DEVICE	PWM_ENABLE / PWM_DISABLE	None	Enables or disables PWM generator and enables PWM pins.
PWM_CLEAR_RELOAD_FLAG	NULL	None	Clears the PWM Reload Interrupt Flag.
PWM_LOAD_OK	NULL	None	Causes loading of the buffered PWM registers to take effect at the next PWM reload.
PWM_FAULT_INT_ENABLE	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Enables corresponding FAULTx CPU interrupt request.
PWM_FAULT_INT_DISABLE	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Disables corresponding FAULTx CPU interrupt request.
PWM_SET_AUTOMATIC_FAULT_CLEAR	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Sets the automatic fault clearing of FAULTx pin fault.
PWM_SET_MANUAL_FAULT_CLEAR	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Sets the manual fault clearing of FAULTx pin fault.
PWM_CLEAR_FAULT_FLAG	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Clears corresponding FAULTx Pin Flag.
PWM_OUTPUT_PAD	PWM_ENABLE / PWM_DISABLE	None	Output Pad enable or disable.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_OUTPUT_SOFTWARE_ CONTROL	pwm_tPWMSignalMask	None	Enables software control of the selected PWM pins. **+**
PWM_OUTPUT_CONTROL	pwm_tPWMSignalMask	None	Controls the state of the corresponding PWM pins. **+**
PWM_SET_MODULO	UWord16	None	Sets the PWM period in PWM clock periods.
PWM_GET_MODULO	NULL	UWord16	Returns the content of PWM Counter Modulo Register.
only on MC56F83xx: PWM_SET_DEADTIME	UWord16	None	Sets the number of IP Bus clock cycles as a deadtime in complementary channel operation mode.
only on MC56F80xx: PWM_SET_DEADTIME_0	UWord16	None	Sets the number of IP Bus clock cycles as a deadtime in complementary channel operation mode. It controls the deadtime during 0 to 1 transitions of the PWM output and during 1 to 0 transitions of the complementary output.
only on MC56F80xx: PWM_SET_DEADTIME_1	UWord16	None	Sets the number of IP Bus clock cycles as a deadtime in complementary channel operation mode. It controls the deadtime during 1 to 0 transitions of the primary output and 0 to 1 transitions of the complementary output.
PWM_WRITE_DISABLE_MAPPING_ REG1	UWord16	None	Determines which PWM pins are disabled by the fault detection decoder (INDISMAP1 register).
PWM_WRITE_DISABLE_MAPPING_ REG2	UWord16	None	Determines which PWM pins are disabled by the fault detection decoder (INDISMAP2 register).
PWM_SET_ALIGNMENT	PWM_EDGE / PWM_CENTER	None	Determines whether all PWM channels will use edge- or center-aligned waveforms.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_SET_NEG_TOP_SIDE_ POLARITY	(PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01) / PWM_ZERO_MASK	None	Determines the polarity for the top-side PWMs. The negative top-side polarity is set for the selected channels (the rest of channels uses the positive top-side polarity).
PWM_SET_NEG_BOTTOM_SIDE_ POLARITY	(PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01) / PWM_ZERO_MASK	None	Determines the polarity for the bottom-side PWMs. The negative bottom-side polarity is set for the selected channels (the rest of channels uses the positive bottom-side polarity).
PWM_SET_INDEPENDENT_ OPERATION	(PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01) / PWM_ZERO_MASK	None	Determines whether the PWM channels will be independent PWMs or complementary PWM pairs. The independent operation is set for the selected channels (the rest of channels are configured as complementary pairs). **+**
PWM_SET_COMPLEMENTARY_ MODE	PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01	None	Sets the selected channels to be complementary PWM pairs.
PWM_SET_INDEPENDENT_ MODE	PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01	None	Sets the selected channels to be independent PWMs.
PWM_SET_WRITE_PROTECT	NULL	None	Write-protects the registers where write protection applies.
PWM_HARDWARE_ACCELERATION	PWM_ENABLE / PWM_DISABLE	None	Enables or disables the hardware accelerator feature.
PWM_SET_CHANNEL_MASK	pwm_tPWMSignalMask	None	Masks the selected PWM logical channels (the rest of channels are unmasked). This command should be used in the independent mode. In complementary mode the masked channels are set to 0% duty cycle and the corresponding complementary channels are therefore set to 100% duty cycle. **+**

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_SET_LOAD_MODE	PWM_LOAD_INDEP / PWM_LOAD_FROM_0_TO_5 / PWM_LOAD_FROM_0_TO_3	None	Sets the way the PWM Value Registers are being loaded.
PWM_SET_SWAP	(PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01) / PWM_ZERO_MASK	None	Swaps the selected channels. +
PWM_WRITE_VALUE_REG_0	Word16	None	Writes the new value to the PWM Value Register 0.
PWM_WRITE_VALUE_REG_1	Word16	None	Writes the new value to the PWM Value Register 1.
PWM_WRITE_VALUE_REG_2	Word16	None	Writes the new value to the PWM Value Register 2.
PWM_WRITE_VALUE_REG_3	Word16	None	Writes the new value to the PWM Value Register 3.
PWM_WRITE_VALUE_REG_4	Word16	None	Writes the new value to the PWM Value Register 4.
PWM_WRITE_VALUE_REG_5	Word16	None	Writes the new value to the PWM Value Register 5.
PWM_WRITE_VALUE_REGS_COMPL	pwm_sComplementaryValues*	None	Writes the new value to the PWM Value Register 0, 2 & 4.
PWM_WRITE_VALUE_REGS_INDEP	pwm_sIndependentValues*	None	Writes the new value to the PWM Value Register 0, 1, 2, 3, 4 & 5.
PWM_READ_FAULT_STATUS_REG	NULL	UWord16	Returns the content of PWM Fault Status & Acknowledge Register.
PWM_READ_COUNTER_REG	NULL	UWord16	Returns the content of PWM Counter Register.
PWM_READ_CONTROL_REG	NULL	UWord16	Returns the content of PWM Control Register.
PWM_READ_PORT_REG	NULL	UWord16	Returns the content of PWM Port Register.
PWM_GET_CURRENT_STATUS_ INPUTS	NULL	UWord16	Returns values of the three current status inputs.
PWM_GET_FAULT_INPUTS	NULL	UWord16	Returns values of the four fault inputs.
PWM_GET_FAULT_INPUT_0	NULL	UWord16	Returns value of the fault input 0.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_GET_FAULT_INPUT_1	NULL	UWord16	Returns value of the fault input 1.
PWM_GET_FAULT_INPUT_2	NULL	UWord16	Returns value of the fault input 2.
PWM_GET_FAULT_INPUT_3	NULL	UWord16	Returns value of the fault input 3.
PWM_SOFTWARE_OUTPUTS_ CONTROL	pwm_sOutputControl*	None	Sets the desired PWM output pins to be controlled by software and activates/deactivates the PWM outputs. +
PWM_UPDATE_VALUE_REG_0	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 0. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REG_1	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 1. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REG_2	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 2. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REG_3	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 3. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REG_4	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 4. The LDOK bit is set afterwards.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_UPDATE_VALUE_REG_5	Word16	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 5. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REGS_ COMPL	pwm_sComplementaryValues*	None	Recalculates the input value of duty cycles (in percentage) with respect to the PWM modulus and writes the resulting values to the three corresponding PWM Value Registers. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_REGS_ INDEP	pwm_sIndependentValues*	None	Recalculates the input value of duty cycles (in percentage) with respect to the PWM modulus and writes the resulting values to the six corresponding PWM Value Registers. The LDOK bit is set afterwards.
PWM_UPDATE_VALUE_SET_ VLMODE	pwm_sUpdateValueSetVlmode*	None	Recalculates the input value of duty cycle (in percentage) with respect to the PWM modulus and writes the resulting value to the PWM Value Register 0 and sets the desired Value Register Load Mode. The LDOK bit is set afterwards. **+**
PWM_SET_MASK_SWAP	pwm_sChannelControl*	None	Masks the selected PWM logical channels and sets the desired swapping channel operation. This command should be used in the independent mode. In complementary mode the masked channels are set to 0% duty cycle and the corresponding complementary channels are therefore set to 100% duty cycle.
PWM_DEBUG_OPERATION	PWM_STOP / PWM_RUN	None	Enables or disables PWM operation during debug mode.
PWM_WAIT_OPERATION	PWM_STOP / PWM_RUN	None	Enables or disables PWM operation during WAIT mode.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
PWM_MASK_SWAP_OPERATION	PWM_80X_ COMPATIBLE / PWM_ENHANCED	None	Determines the functionality of the mask & swap operations.
PWM_SET_HALF_CYCLE_INTERNAL_ CORRECTION	PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01	None	Sets half cycle internal correction method.
PWM_SET_FULL_CYCLE_INTERNAL_ CORRECTION	PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01	None	Sets full cycle internal correction method.
PWM_READ_INTERNAL_CORRECTION_ CONTROL_REG	NULL	UWord16	Returns the content of PWM Internal Correction Control Register.
only on MC56F80xx: PWM_SET_SOURCE_0	PWM_SOURCE_PWM / PWM_SOURCE_ADC_0 / PWM_SOURCE_PWMSRC_0 / PWM_SOURCE_TMR_0	None	Controls bits to determine the source signal for the complementary PWM0/PWM1 pair output.
only on MC56F80xx: PWM_SET_SOURCE_1	PWM_SOURCE_PWM / PWM_SOURCE_ADC_1 / PWM_SOURCE_PWMSRC_1 / PWM_SOURCE_TMR_1 / PWM_SOURCE_SRC_0	None	Controls bits to determine the source signal for the complementary PWM2/PWM3 pair output.
only on MC56F80xx: PWM_SET_SOURCE_2	PWM_SOURCE_PWM / PWM_SOURCE_ADC_2 / PWM_SOURCE_PWMSRC_2 / PWM_SOURCE_TMR_2 / PWM_SOURCE_SRC_0	None	Controls bits to determine the source signal for the complementary PWM4/PWM5 pair output.
only on MC56F80xx: PWM_SET_COMPARE_INVERT_0 PWM_SET_COMPARE_INVERT_1 PWM_SET_COMPARE_INVERT_2 PWM_SET_COMPARE_INVERT_3 PWM_SET_COMPARE_INVERT_4 PWM_SET_COMPARE_INVERT_5	PWM_NORMAL / PWM_INVERT	None	Controls the polarity of the PWM outputs 0-5.
only on MC56F802x/3x and MC56F800x: PWM_SET_ACTIVE_HIGH_FAULTS	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Configures selected fault inputs as active-high.
only on MC56F802x/3x and MC56F800x: PWM_SET_ACTIVE_LOW_FAULTS	PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3	None	Configures selected fault inputs as active-low.
only on MC56F802x/3x and MC56F800x: PWM_DISABLE_SYNC	NULL	None	Disables SYNC input/output feature, reverts pin to FAULT2 operation.
only on MC56F802x/3x and MC56F800x: PWM_ENABLE_SYNC_OUT	NULL	None	Enables SYNC signal output on FAULT2 pin.

Table 5-453. ioctl commands (Continued)

cmd	param	Return	Description
only on MC56F802x/3x and MC56F800x: PWM_ENABLE_SYNC_IN	UWord16 in range 1..0x7fff	None	Enables SYNC input in time window of a given size.
only on MC56F802x/3x and MC56F800x: PWM_READ_FILT0_REG PWM_READ_FILT1_REG PWM_READ_FILT2_REG PWM_READ_FILT3_REG	NULL	UWord16	Returns a content of the FAULT Filter Registers 0-3.
only on MC56F802x/3x and MC56F800x: PWM_WRITE_FILT0_REG PWM_WRITE_FILT1_REG PWM_WRITE_FILT2_REG PWM_WRITE_FILT3_REG	UWord16	None	Writes given value to one of the FAULT Filter Registers.
only on MC56F800x: PWM_SET_PULSE_EDGE_CONTROL_0 PWM_SET_PULSE_EDGE_CONTROL_1 PWM_SET_PULSE_EDGE_CONTROL_2	PWM_NORMAL_MODE / PWM_PWMVAL_MODE	None	Sets Pulse edge control

+ Note - ensure that this *ioctl* call cannot be interrupted.

5.14.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.14.3.1 PWM_INIT - initialize PWM module

Call(s):

```
void ioctl(const int *pModuleBase, PWM_INIT, NULL);
```

Arguments:

Table 5-454. PWM_INIT ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_INIT* ioctl command is used to initialize the selected PWM module. Initialization consists of writing the initialization values to the following PWM registers: PWM Control Register (PMCTL), PWM Fault Control Register (PMFCTL), PWM Output Control Register (PMOUT), PWM Counter Modulo Register (PWMCM), PWM Deadtime Register (PMDAEDTM) on MC56F83xx or PWM Deadtime Registers 0-1 (DTIM0-1) on MC56F80xx, PWM Disable Mapping Registers 1-2 (PMDISMAP1-2), PWM Configure Register (PMCFG), PWM Channel Control Register (PMCCR), PWM Internal Correction Register (PMICCR) and PWM Source Control Register (SCTRL). On MC56F80xx the following registers are further configured: PWM Synchronization Window Register (SYNC) and PWM Fault Filter Registers (FFILT0-3). Initialization values for each register are taken from the *appconfig.h* file, where each configuration item corresponds to one PWM register. If no initialization value is defined in *appconfig.h*, then no initialization of particular register is performed. For reference on symbols of configuration items, see Section 5.14.2.3.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_INIT* ioctl command is implemented as a function.

Example 5-402. PWM_INIT

```
ioctl(PWM_A, PWM_INIT, NULL);
```

This code initializes the PWM A module by the initialization values taken from *appconfig.h*.

```
ioctl(PWM_B, PWM_INIT, NULL);
```

This code initializes the PWM B module by the initialization values taken from *appconfig.h*.

5.14.3.2 PWM_SET_RELOAD_FREQUENCY - set PWM reload frequency

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_RELOAD_FREQUENCY,
           UWord16 param);
```

Arguments:

Table 5-455. PWM_SET_RELOAD_FREQUENCY ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired PWM load frequency. Use one of the predefined constants: PWM_RELOAD_OPPORTUNITY_1 / PWM_RELOAD_OPPORTUNITY_2 / PWM_RELOAD_OPPORTUNITY_3 / PWM_RELOAD_OPPORTUNITY_4 / PWM_RELOAD_OPPORTUNITY_5 / PWM_RELOAD_OPPORTUNITY_6 / PWM_RELOAD_OPPORTUNITY_7 / PWM_RELOAD_OPPORTUNITY_8 / PWM_RELOAD_OPPORTUNITY_9 / PWM_RELOAD_OPPORTUNITY_10 / PWM_RELOAD_OPPORTUNITY_11 / PWM_RELOAD_OPPORTUNITY_12 / PWM_RELOAD_OPPORTUNITY_13 / PWM_RELOAD_OPPORTUNITY_14 / PWM_RELOAD_OPPORTUNITY_15 / PWM_RELOAD_OPPORTUNITY_16

Description: The *PWM_SET_RELOAD_FREQUENCY* ioctl command selects the PWM load frequency by selecting a certain order of opportunity. This command manipulates the LDFQ bits (Bit 15 - 12) in the PWM Control Register.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_SET_RELOAD_FREQUENCY* ioctl command is implemented as a macro.

Example 5-403. PWM_SET_RELOAD_FREQUENCY

```
ioctl(PWM_A, SET_RELOAD_FREQUENCY, PWM_RELOAD_OPPORTUNITY_2);
```

This code sets the PWM A load frequency to every 2 PWM opportunities.

5.14.3.3 PWM_HALF_CYCLE_RELOAD - enable or disable half-cycle reload

Call(s):

```
void ioctl(const int *pModuleBase, PWM_HALF_CYCLE_RELOAD,
           UWord16 param);
```

Arguments:

Table 5-456. PWM_HALF_CYCLE_RELOAD ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired action. Use PWM_ENABLE to enable half-cycle reload or PWM_DISABLE to disable it.

Description: The *PWM_HALF_CYCLE_RELOAD* ioctl command enables or disables half-cycle reloads in center-aligned PWM operation mode. This command sets the HALF bit (Bit 11) in the PWM Control Register when PWM_ENABLE is used as a parameter. This command clears the above mentioned bit in case of PWM_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: This command has no effect on edge-aligned PWMs (hardware feature). Use of this ioctl command may unintentionally result in clearing an interrupt flag

Design/Implementation: The *PWM_HALF_CYCLE_RELOAD* ioctl command is implemented as a macro.

Example 5-404. PWM_HALF_CYCLE_RELOAD

```
ioctl(PWM_A, PWM_HALF_CYCLE_RELOAD, PWM_ENABLE);
```

This code enables half-cycle reload for the PWM A module.

```
ioctl(PWM_B, PWM_HALF_CYCLE_RELOAD, PWM_DISABLE);
```

This code disables half-cycle reload for the PWM B module.

5.14.3.4 PWM_SET_CURRENT_POLARITY - set the current polarity bits

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_CURRENT_POLARITY,
           UWord16 param);
```

Arguments:

Table 5-457. PWM_SET_CURRENT_POLARITY ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to set the selected current polarity bits. Use combination of the predefined constants: PWM_IPOL_0 | PWM_IPOL_1 | PWM_IPOL_2 Use PWM_ZERO_MASK to clear all current polarity bits.

Description: The *PWM_SET_CURRENT_POLARITY* ioctl command selects the desired PWM Value Register for the top/bottom software correction. The PWM_IPOL_0 selects PWM Value Register 1 for the PWM0 and PWM1 pins in top/bottom correction, otherwise PWM Value Register 0 is used. The PWM_IPOL_1 selects PWM Value Register 3 for the PWM2 and PWM3 pins in top/bottom correction, otherwise PWM Value Register 2 is used. The PWM_IPOL_2 selects PWM Value Register 5 for the PWM4 and PWM5 pins in top/bottom correction, otherwise PWM Value Register 4 is used. This command manipulates the IPOL0 bit (Bit 8), IPOL1 bit (Bit 9) and IPOL2 bit (Bit 10) in the PWM Control Register.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_SET_CURRENT_POLARITY* ioctl command is implemented as a macro.

Example 5-405. PWM_SET_CURRENT_POLARITY

```
ioctl(PWM_A, PWM_SET_CURRENT_POLARITY, PWM_IPOL_0);
```

This code selects the PWM A module Value Register 1 for the PWM0 and PWM1 pins in top/bottom correction.

```
ioctl(PWM_B, PWM_SET_CURRENT_POLARITY, PWM_IPOL_1 | PWM_IPOL_2);
```

This code selects the PWM B module Value Register 3 for the PWM2 and PWM3 pins in top/bottom correction, and the PWM Value Register 5 for the PWM4 and PWM5 pins in top/bottom correction.

5.14.3.5 PWM_SET_PRESCALER - set the PWM clock frequency

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-458. PWM_SET_PRESCALER ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the PWM clock frequency. Use one of the predefined constants: PWM_PRESCALER_DIV_1 / PWM_PRESCALER_DIV_2 / PWM_PRESCALER_DIV_4 / PWM_PRESCALER_DIV_8

Description: The *PWM_SET_PRESCALER* ioctl command sets the PWM frequency as a fraction of the IP Bus frequency, see Table 5-459. This command manipulates the PRSC bits (Bit 7-6) in the PWM Control Register.

Table 5-459. PWM Prescaler

param	PWM clock frequency
PWM_PRESCALER_DIV_1	IPBus
PWM_PRESCALER_DIV_2	IPBus / 2
PWM_PRESCALER_DIV_4	IPBus / 4
PWM_PRESCALER_DIV_8	IPBus / 8

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_SET_PRESCALER* ioctl command is implemented as a macro.

Example 5-406. PWM_SET_PRESCALER

```
ioctl(PWM_A, PWM_SET_PRESCALER, PWM_PRESCALER_DIV_2);
```

This code sets the PWM A clock frequency as one half of the IP Bus frequency.

```
ioctl(PWM_B, PWM_SET_PRESCALER, PWM_PRESCALER_DIV_1);
```

This code sets the PWM B clock frequency equal to the IP Bus frequency.

5.14.3.6 PWM_RELOAD_INT - enable or disable PWM reload interrupt

Call(s):

```
void ioctl(const int *pModuleBase, PWM_RELOAD_INT, UWord16 param);
```

Arguments:

Table 5-460. PWM_RELOAD_INT ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired action. Use PWM_ENABLE to enable the PWM reload interrupt request or PWM_DISABLE to disable it.

Description: The *PWM_RELOAD_INT* ioctl command enables or disables PWM reload interrupt requests. This command sets the PWMRIE bit (Bit 5) in the PWM Control Register when PWM_ENABLE is used as a parameter. This command clears the above mentioned bit in case of PWM_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_RELOAD_INT* ioctl command is implemented as a macro.

Example 5-407. PWM_RELOAD_INT

```
ioctl(PWM_A, PWM_RELOAD_INT, PWM_ENABLE);
```

This code enables a PWM A reload interrupt request.

```
ioctl(PWM_B, PWM_RELOAD_INT, PWM_DISABLE);
```

This code disables any PWM B reload interrupt request.

5.14.3.7 PWM_SET_CURRENT_SENSING - set the correction scheme

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_CURRENT_SENSING,
           UWord16 param);
```

Arguments:

Table 5-461. PWM_SET_CURRENT_SENSING ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired top/bottom correction scheme. Use one of the predefined constants: PWM_CORRECTION_NO / PWM_CORRECTION_SOFTWARE / PWM_CORRECTION_DURING_DEADTIME / PWM_CORRECTION_DURING_CYCLE

Description: The *PWM_SET_CURRENT_SENSING* ioctl command selects the top/bottom correction scheme, see Table 5-462. This command manipulates the ISENS bits (Bit 3-2) in the PWM Control Register.

Table 5-462. Correction Method Selection

param	Correction Method
PWM_CORRECTION_NO	no correction
PWM_CORRECTION_SOFTWARE	manual - software correction
PWM_CORRECTION_DURING_DEADTIME	current status sample correction on pins IS1, IS2 and IS3 during deadtime
PWM_CORRECTION_DURING_CYCLE	current status sample correction on pins IS1, IS2 and IS3 at the half-cycle in center-aligned operation or at the end of the cycle in edge-aligned operation

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_SET_CURRENT_SENSING* ioctl command is implemented as a macro.

Example 5-408. PWM_SET_CURRENT_SENSING

```
ioctl(PWM_B,PWM_SET_CURRENT_SENSING,PWM_CORRECTION_DURING_CYCLE);
```

This code sets the PWM B sample correction on pins IS1, IS2 and IS3 at the half-cycle in center-aligned operation or at the end of the cycle in edge-aligned operation.

5.14.3.8 PWM_DEVICE - enable or disable PWM generator

Call(s):

```
void ioctl(const int *pModuleBase, PWM_DEVICE, UWord16 param);
```

Arguments:

Table 5-463. PWM_DEVICE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired action. Use PWM_ENABLE to enable the PWM generator or PWM_DISABLE to disable it.

Description: The *PWM_DEVICE* ioctl command enables or disables the PWM generator and the PWM pins. This command sets the PWMEN bit (Bit 0) in the PWM Control Register when PWM_ENABLE is used as a parameter. This command clears the above mentioned bit in case of PWM_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_DEVICE* ioctl command is implemented as a macro.

Example 5-409. PWM_DEVICE

```
ioctl(PWM_A, PWM_DEVICE, PWM_ENABLE);
```

This code enables the PWM A generator and the corresponding pins.

```
ioctl(PWM_B, PWM_DEVICE, PWM_DISABLE);
```

This code disables the PWM B generator and the corresponding pins.

5.14.3.9 PWM_CLEAR_RELOAD_FLAG - clear PWM reload interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, PWM_CLEAR_RELOAD_FLAG, NULL);
```

Arguments:

Table 5-464. PWM_CLEAR_RELOAD_FLAG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_CLEAR_RELOAD_FLAG* ioctl command clears the PWM reload flag. This command clears the PWMF bit (Bit 4) in the PWM Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_CLEAR_RELOAD_FLAG* ioctl command is implemented as a macro.

Example 5-410. PWM_CLEAR_RELOAD_FLAG

```
ioctl(PWM_A, PWM_CLEAR_RELOAD_FLAG, NULL);
```

This code clears the PWM A reload flag.

5.14.3.10 PWM_LOAD_OK - load the new effective values into PWM registers

Call(s):

```
void ioctl(const int *pModuleBase, PWM_LOAD_OK, NULL);
```

Arguments:

Table 5-465. PWM_LOAD_OK ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_LOAD_OK* ioctl command loads the prescaler bits (PRSC) of the PWM Control Register, the PWM Modulo Register and the PWM Value Registers into a set of buffers, to take effect at the next PWM Reload. This command sets the LDOK bit (Bit 1) in the PWM Control Register.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt flag.

Design/Implementation: The *PWM_LOAD_OK* ioctl command is implemented as a macro.

Example 5-411. PWM_LOAD_OK

```
ioctl(PWM_A, PWM_LOAD_OK, NULL);
```

This code loads prescaler, modulus and PWM values of PWM A.

5.14.3.11 PWM_FAULT_INT_ENABLE - enable FAULT interrupt requests

Call(s):

```
void ioctl(const int *pModuleBase, PWM_FAULT_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-466. PWM_FAULT_INT_ENABLE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired fault interrupt request. Use one of the pre-defined constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_FAULT_INT_ENABLE* ioctl command enables FAULTx Pin interrupt requests. This command sets the FIEx bits (Bit 7, 5, 3, 1) in the PWM Fault Control Register.

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_FAULT_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-412. PWM_FAULT_INT_ENABLE

```
ioctl(PWM_A, PWM_FAULT_INT_ENABLE, PWM_FAULT_1);
```

This code enables a PWM A FAULT1 Pin interrupt request.

```
ioctl(PWM_B, PWM_FAULT_INT_ENABLE, PWM_FAULT_3);
```

This code enables a PWM B FAULT3 Pin interrupt request.

5.14.3.12 PWM_FAULT_INT_DISABLE - disable FAULT interrupt requests

Call(s):

```
void ioctl(const int *pModuleBase, PWM_FAULT_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-467. PWM_FAULT_INT_DISABLE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired fault interrupt request. Use one of the pre-defined constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_FAULT_INT_DISABLE* ioctl command disables FAULTx Pin interrupt requests. This command clears the FIEx bits (Bit 7, 5, 3, 1) in the PWM Fault Control Register.

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_FAULT_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-413. PWM_FAULT_INT_DISABLE

```
ioctl(PWM_A, PWM_FAULT_INT_DISABLE, PWM_FAULT_1);
```

This code disables any PWM A FAULT1 Pin interrupt request.

```
ioctl(PWM_B, PWM_FAULT_INT_DISABLE, PWM_FAULT_0);
```

This code disables a PWM B FAULT0 Pin interrupt request.

5.14.3.13 PWM_SET_AUTOMATIC_FAULT_CLEAR - set automatic FAULT clearing

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_AUTOMATIC_FAULT_CLEAR,
           UWord16 param);
```

Arguments:

Table 5-468. PWM_SET_AUTOMATIC_FAULT_CLEAR ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired automatic fault clearing. Use one of the predefined constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_AUTOMATIC_FAULT_CLEAR* ioctl command sets automatic clearing of FAULTx Pin faults. This command sets the FMODEx bits (Bit 6, 4, 2, 0) in the PWM Fault Control Register.

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_AUTOMATIC_FAULT_CLEAR* ioctl command is implemented as a macro.

Example 5-414. PWM_SET_AUTOMATIC_FAULT_CLEAR

```
ioctl(PWM_A, PWM_SET_AUTOMATIC_FAULT_CLEAR, PWM_FAULT_1);
```

This code sets automatic clearing of a FAULT1 Pin fault on PWM A.

```
ioctl(PWM_B, PWM_SET_AUTOMATIC_FAULT_CLEAR, PWM_FAULT_0);
```

This code sets automatic clearing of FAULT0 Pin fault on PWM B.

5.14.3.14 PWM_SET_MANUAL_FAULT_CLEAR - set manual FAULT clearing

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_MANUAL_FAULT_CLEAR,
           UWord16 param);
```

Arguments:

Table 5-469. PWM_SET_MANUAL_FAULT_CLEAR ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired manual fault clearing. Use one of the pre-defined constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_MANUAL_FAULT_CLEAR* ioctl command sets manual clearing of FAULTx Pin faults. This command clears the FMODEx bits (Bit 6, 4, 2, 0) in the PWM Fault Control Register.

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_MANUAL_FAULT_CLEAR* ioctl command is implemented as a macro.

Example 5-415. PWM_SET_MANUAL_FAULT_CLEAR

```
ioctl(PWM_A, PWM_SET_MANUAL_FAULT_CLEAR, PWM_FAULT_1);
```

This code sets manual clearing of a FAULT1 Pin fault on PWM A.

```
ioctl(PWM_B, PWM_SET_MANUAL_FAULT_CLEAR, PWM_FAULT_3);
```

This code sets manual clearing of FAULT3 Pin fault on PWM B.

5.14.3.15 PWM_CLEAR_FAULT_FLAG - clear FAULT flag

Call(s):

```
void ioctl(const int *pModuleBase, PWM_CLEAR_FAULT_FLAG,
           UWord16 param);
```

Arguments:

Table 5-470. PWM_CLEAR_FAULT_FLAG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the fault pin flag to clear. Use one of the predefined constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_CLEAR_FAULT_FLAG* ioctl command clears FAULTx Pin fault flags, i.e. FFLAGx bits (Bit 14, 12, 10, 8) by writing a logic one to the FTACKx bits (Bit 6, 4, 2, 0) in the PWM Fault Status & Acknowledge Register.

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_CLEAR_FAULT_FLAG* ioctl command is implemented as a macro.

Example 5-416. PWM_CLEAR_FAULT_FLAG

```
ioctl(PWM_A, PWM_CLEAR_FAULT_FLAG, PWM_FAULT_1);
```

This code clears the FAULT1 Pin fault flag on PWM A.

```
ioctl(PWM_B, PWM_CLEAR_FAULT_FLAG, PWM_FAULT_0);
```

This code clears FAULT0 Pin fault flag on PWM B.

5.14.3.16 PWM_OUTPUT_PAD - enable or disable PWM output pads

Call(s):

```
void ioctl(const int *pModuleBase, PWM_OUTPUT_PAD, UWord16 param);
```

Arguments:

Table 5-471. PWM_OUTPUT_PAD ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired action. Use PWM_ENABLE to enable the PWM output pads or PWM_DISABLE to disable them.

Description: The *PWM_OUTPUT_PAD* ioctl command enables or disables PWM output pads. This command sets the PAD_EN bit (Bit 15) in the PWM Output Control Register when PWM_ENABLE is used as a parameter. This command clears the above mentioned bit in case of PWM_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_OUTPUT_PAD* ioctl command is implemented as a macro.

Example 5-417. PWM_OUTPUT_PAD

```
ioctl(PWM_A, PWM_OUTPUT_PAD, PWM_ENABLE);
```

This code enables the PWM A output pads.

```
ioctl(PWM_B, PWM_OUTPUT_PAD, PWM_DISABLE);
```

This code disables the PWM B output pads.

5.14.3.17 PWM_OUTPUT_SOFTWARE_CONTROL - enable software control of PWM pins

Call(s):

```
void ioctl(const int *pModuleBase, PWM_OUTPUT_SOFTWARE_CONTROL,
           pwm_tPWMSignalMask param);
```

Arguments:

Table 5-472. PWM_OUTPUT_SOFTWARE_CONTROL ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the PWM signal. Parameter is of *pwm_tPWMSignalMask* type and contains six control bits representing six PWM signals.

Description: The *PWM_OUTPUT_SOFTWARE_CONTROL* ioctl command enables software control of the corresponding PWM pins. The PWM pins/signals, specified by *param*, can be controlled by software, while the remaining pins cannot be controlled by software. This command sets the OUTCTL bits (Bit 13-8) in the PWM Output Control Register.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.

Design/Implementation: The *PWM_OUTPUT_SOFTWARE_CONTROL* ioctl command is implemented as a macro.

Example 5-418. PWM_OUTPUT_SOFTWARE_CONTROL

```
pwm_tPWMSignalMask tempSignals;

tempSignals = MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_5;

ioctl(PWM_A, PWM_OUTPUT_SOFTWARE_CONTROL, tempSignals);
```

This code enables PWM2 and PWM5 pins of PWM A to be controlled by software.

```
ioctl(PWM_B, PWM_OUTPUT_SOFTWARE_CONTROL, \
           MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_5);
```

This code enables PWM2 and PWM5 pins of PWM B to be controlled by software.

```
ioctl(PWM_B, PWM_OUTPUT_SOFTWARE_CONTROL, MC_PWM_NO_SIGNALS);
```

This code disables all PWM pins of PWM B to be controlled by software.

5.14.3.18 PWM_OUTPUT_CONTROL - control the state of PWM pins

Call(s):

```
void ioctl(const int *pModuleBase, PWM_OUTPUT_CONTROL,
           pwm_tPWMSignalMask param);
```

Arguments:

Table 5-473. PWM_OUTPUT_CONTROL ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to control the PWM signal. Parameter is of *pwm_tPWMSignalMask* type and contains six control bits representing six PWM signals.

Description: The *PWM_OUTPUT_CONTROL* ioctl command controls the state of the PWM pins when the corresponding PWM pins are enabled to be controlled by software (see Section 5.14.3.17). See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details. This command sets the OUT bits (Bit 5-0) in the PWM Output Control Register.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.

Design/Implementation: The *PWM_OUTPUT_CONTROL* ioctl command is implemented as a macro.

Example 5-419. PWM_OUTPUT_CONTROL

```
pwm_tPWMSignalMask tempSignals;

tempSignals = MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_4;

ioctl(PWM_A, PWM_OUTPUT_CONTROL, tempSignals);
```

This code activates PWM2 and PWM4 pins of PWM A.

```
ioctl(PWM_B, PWM_OUTPUT_CONTROL,
           MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_4);
```

This code activates PWM2 and PWM4 pins of PWM B.

5.14.3.19 PWM_SET_MODULO - set the PWM period

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_MODULO, UWord16 param);
```

Arguments:

Table 5-474. PWM_SET_MODULO ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value representing PWM modulus.

Description: The *PWM_SET_MODULO* ioctl command writes the value (*param*) to the PWM Counter Modulo Register (PWMCM). This value represents the PWM period expressed in PWM clock periods.

Returns: None.

Range Issues: *param* must be a 15-bit unsigned value.

Special Issues: None.

Design/Implementation: The *PWM_SET_MODULO* ioctl command is implemented as a macro.

Example 5-420. PWM_SET_MODULO

```
UWord16          modulo;

modulo = 0x7fff;

ioctl(PWM_A, PWM_SET_MODULO, modulo);
```

This code writes a new value to the PWM A Counter Modulo Register.

```
ioctl(PWM_B, PWM_SET_MODULO, 0x0fff);
```

This code writes 0x0fff to the PWM B Counter Modulo Register.

5.14.3.20 PWM_GET_MODULO - return the content of the PWM Modulo Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_GET_MODULO, NULL);
```

Arguments:

Table 5-475. PWM_GET_MODULO ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_GET_MODULO* ioctl command returns the content of the PWM Counter Modulo Register (PWMCM).

Returns: the content of the PWM Modulo Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_GET_MODULO* ioctl command is implemented as a macro.

Example 5-421. PWM_GET_MODULO

```
UWord16          modulo;

modulo = ioctl(PWM_A, PWM_GET_MODULO, NULL);
```

This code stores the content of the PWM A Counter Modulo Register to the modulo variable.

```
modulo = ioctl(PWM_B, PWM_GET_MODULO, NULL);
```

This code stores the content of the PWM B Counter Modulo Register to the modulo variable.

5.14.3.21 PWM_SET_DEADTIME - set deadtime for complementary channel operation mode

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_DEADTIME,
           UWord16 param);
```

Arguments:

Table 5-476. PWM_SET_DEADTIME ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value representing deadtime.

Description: The *PWM_SET_DEADTIME* ioctl command writes the value (*param*) to the PWM Deadtime Register (PMDEADTM). This value represents the number of PWM clock cycles in complementary channel operation.

Returns: None.

Range Issues: *param* must be within <0x000, 0x0fff>.

Special Issues: This command is primarily intended to be used on MC56F83xx. It writes the same value to both PWM Deadtime Registers (DTIM0-1) on MC56F80xx.

Design/Implementation: The *PWM_SET_DEADTIME* ioctl command is implemented as a macro.

Example 5-422. PWM_SET_DEADTIME

```
UWord16          deadtime;

deadtime = 0x00ff;

ioctl(PWM_A, PWM_SET_DEADTIME, deadtime);
```

This code writes a new value to the PWM A Deadtime Register.

```
ioctl(PWM_B, PWM_SET_DEADTIME, 0x007f);
```

This code writes 0x007f to the PWM B Deadtime Register.

5.14.3.22 PWM_SET_DEADTIME_0 - set deadtime during 0 to 1 transitions of the PWM output and during 1 to 0 transitions of the complementary output

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_DEADTIME_0,
           UWord16 param);
```

Arguments:

Table 5-477. PWM_SET_DEADTIME_0 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	Value representing deadtime.

Description: The *PWM_SET_DEADTIME_0* ioctl command writes the value (*param*) to the PWM Deadtime Register 0 (DTIM0). This value represents the number of PWM clock cycles in complementary channel operation.

Returns: None.

Range Issues: *param* must be within <0x000, 0x0fff>.

Special Issues: This command is applicable only on MC56F80xx.

Design/Implementation: The *PWM_SET_DEADTIME_0* ioctl command is implemented as a macro.

Example 5-423. PWM_SET_DEADTIME_0

```
UWord16          deadtime;

deadtime = 0x00ff;

ioctl(PWM_A, PWM_SET_DEADTIME_0, deadtime);
```

This code writes a new value to the PWM A Deadtime Register 0.

```
ioctl(PWM_B, PWM_SET_DEADTIME_0, 0x007f);
```

This code writes 0x007f to the PWM B Deadtime Register 0.

5.14.3.23 PWM_SET_DEADTIME_1 - set deadtime during 1 to 0 transitions of the PWM output and during 0 to 1 transitions of the complementary output

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_DEADTIME_1,
           UWord16 param);
```

Arguments:

Table 5-478. PWM_SET_DEADTIME_1 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	Value representing deadtime.

Description: The *PWM_SET_DEADTIME_1* ioctl command writes the value (*param*) to the PWM Deadtime Register 1 (DTIM1). This value represents the number of PWM clock cycles in complementary channel operation.

Returns: None.

Range Issues: *param* must be within <0x000, 0x0fff>.

Special Issues: This command is applicable only on MC56F80xx.

Design/Implementation: The *PWM_SET_DEADTIME_1* ioctl command is implemented as a macro.

Example 5-424. PWM_SET_DEADTIME_1

```
UWord16          deadtime;

deadtime = 0x00ff;

ioctl(PWM_A, PWM_SET_DEADTIME_1, deadtime);
```

This code writes a new value to the PWM A Deadtime Register 1.

```
ioctl(PWM_B, PWM_SET_DEADTIME_1, 0x007f);
```

This code writes 0x007f to the PWM B Deadtime Register 1.

5.14.3.24 PWM_WRITE_DISABLE_MAPPING_REG1 - rewrite the content of the PWM Disable Mapping Register 1

Call(s):

```
void ioctl(const int *pModuleBase,PWM_WRITE_DISABLE_MAPPING_REG1,
           UWord16 param);
```

Arguments:

Table 5-479. PWM_WRITE_DISABLE_MAPPING_REG1 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value to determine disabled PWM pins by the fault protection inputs.

Description: The *PWM_WRITE_DISABLE_MAPPING_REG1* ioctl command writes the value (*param*) to the PWM Disable Mapping Register 1 (PMDISMAP1) in order to disable the required pin-fault combination, according to Table 5-480.

Table 5-480. PWM Pin - Fault Mapping Matrix 1

PWM Pin to be disabled	Fault0	Fault1	Fault2	Fault3	PWM Disable Mapping Register
	to assign Fault and PWM pin set **DISMAP bit #**				
PWM0	0	1	2	3	PMDISMAP1 [0:3]
PWM1	4	5	6	7	PMDISMAP1 [4:7]
PWM2	8	9	10	11	PMDISMAP1 [8:11]
PWM3	12	13	14	15	PMDISMAP1 [12:15]

Returns: None.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_WRITE_DISABLE_MAPPING_REG1* ioctl command is implemented as a macro.

Example 5-425. PWM_WRITE_DISABLE_MAPPING_REG1

```
UWord16 dismap;
dismap = 0x00ff;
ioctl(PWM_A, PWM_WRITE_DISABLE_MAPPING_REG1, dismap);
```

This code writes a new value to the PWM A Disable Mapping Register 1 which disables all faults on PWM2 and PWM3 pins.

```
ioctl(PWM_B, PWM_WRITE_DISABLE_MAPPING_REG1, 0xfff0);
```

This code writes 0xfff0 to the PWM B Disable Mapping Register 1 to disable all faults on PWM0 pin.

5.14.3.25 PWM_WRITE_DISABLE_MAPPING_REG2 - rewrite the content of the PWM Disable Mapping Register 2

Call(s):

```
void ioctl(const int *pModuleBase,PWM_WRITE_DISABLE_MAPPING_REG2,
           UWord16 param);
```

Arguments:

Table 5-481. PWM_WRITE_DISABLE_MAPPING_REG2 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value to determine disabled PWM pins by the fault protection inputs.

Description: The *PWM_WRITE_DISABLE_MAPPING_REG2* ioctl command writes the value (*param*) to the PWM Disable Mapping Register 2 (PMDISMAP2) in order to disable the required pin-fault combination, according to Table 5-482.

Table 5-482. PWM Pin - Fault Mapping Matrix 2

PWM Pin to be disabled	Fault0	Fault1	Fault2	Fault3	PWM Disable Mapping Register
	to assign Fault and PWM pin set **DISMAP bit #**				
PWM4	16	17	18	19	PMDISMAP2 [0:3]
PWM5	20	21	22	23	PMDISMAP2 [4:7]

Returns: None.

Range Issues: *param* must be within <0x000, 0x00ff>.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_WRITE_DISABLE_MAPPING_REG2* ioctl command is implemented as a macro.

Example 5-426. PWM_WRITE_DISABLE_MAPPING_REG2

```
UWord16 dismap;
dismap = 0x00f3;
ioctl(PWM_A, PWM_WRITE_DISABLE_MAPPING_REG2, dismap);
```

This code writes a new value to the PWM A Disable Mapping Register 2 to disable Fault2 and Fault3 on PWM4 pin.

```
ioctl(PWM_B, PWM_WRITE_DISABLE_MAPPING_REG2, PWM_ZERO_MASK);
```

This code writes 0x0000 to the PWM B Disable Mapping Register 2 to disable all faults on PWM4 and PWM5 pins.

5.14.3.26 PWM_SET_ALIGNMENT - set center- or edge-aligned waveforms

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_ALIGNMENT,
           UWord16 param);
```

Arguments:

Table 5-483. PWM_SET_ALIGNMENT ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use PWM_ALIGN_EDGE to set edge-aligned waveforms or PWM_ALIGN_CENTER to set center-aligned waveforms.

Description: The *PWM_SET_ALIGNMENT* ioctl command determines whether all PWM channels will use edge- or center-aligned waveforms. This command sets the EDG bit (Bit 12) in the PWM Configure Register when PWM_ALIGN_EDGE is used as a parameter. This command clears the above mentioned bit in case of PWM_ALIGN_CENTER.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_ALIGNMENT* ioctl command is implemented as a macro.

Example 5-427. PWM_SET_ALIGNMENT

```
ioctl(PWM_A, PWM_SET_ALIGNMENT, PWM_ALIGN_EDGE);
```

This code sets PWM A channels to use edge-aligned waveforms.

```
ioctl(PWM_B, PWM_SET_ALIGNMENT, PWM_ALIGN_CENTER);
```

This code sets PWM B channels to use center-aligned waveforms.

5.14.3.27 PWM_SET_NEG_TOP_SIDE_POLARITY - set the polarity for the top-side PWMs

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_NEG_TOP_SIDE_POLARITY,
           UWord16 param);
```

Arguments:

Table 5-484. PWM_SET_NEG_TOP_SIDE_POLARITY ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01 Use PWM_ZERO_MASK to clear all top-side PWM polarity bits.

Description: The *PWM_SET_NEG_TOP_SIDE_POLARITY* ioctl command determines the polarity of the top-side PWMs. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). The specified channels are set to negative top-side polarity, the remaining ones are set to positive top-side polarity. This command manipulates the TOPNEG bits (Bit 10-8) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_NEG_TOP_SIDE_POLARITY* ioctl command is implemented as a macro.

Example 5-428. PWM_SET_NEG_TOP_SIDE_POLARITY

```
ioctl(PWM_A, PWM_SET_NEG_TOP_SIDE_POLARITY, PWM_CHANNEL_01);
```

This code sets PWM A channel zero to one, to the negative top-side polarity.

```
ioctl(PWM_B, PWM_SET_NEG_TOP_SIDE_POLARITY,
           PWM_CHANNEL_23 | PWM_CHANNEL_45);
```

This code sets PWM B channels two to three and four to five, to the negative top-side polarity.

5.14.3.28 PWM_SET_NEG_BOTTOM_SIDE_POLARITY - set the polarity for the bottom-side PWMs

Call(s):

```
void ioctl(const int *pModuleBase,
           PWM_SET_NEG_BOTTOM_SIDE_POLARITY,
           UWord16 param);
```

Arguments:

Table 5-485. PWM_SET_NEG_BOTTOM_SIDE_POLARITY ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01 Use PWM_ZERO_MASK to clear all bottom-side PWM polarity bits.

Description: The *PWM_SET_NEG_BOTTOM_SIDE_POLARITY* ioctl command determines the polarity of the bottom-side PWMs. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). The specified channels are set to negative bottom-side polarity, the remaining ones are set to positive bottom-side polarity. This command manipulates the BOTNEG bits (Bit 6-4) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_NEG_BOTTOM_SIDE_POLARITY* ioctl command is implemented as a macro.

Example 5-429. PWM_SET_NEG_BOTTOM_SIDE_POLARITY

```
ioctl(PWM_A, PWM_SET_NEG_BOTTOM_SIDE_POLARITY, PWM_CHANNEL_01);
```

This code sets PWM A channel zero to one, to the negative bottom-side polarity.

```
ioctl(PWM_B, PWM_SET_NEG_BOTTOM_SIDE_POLARITY, PWM_ZERO_MASK);
```

This code sets all three PWM B channel to the positive bottom-side polarity.

5.14.3.29 PWM_SET_INDEPENDENT_OPERATION - configure PWM pins as independent PWMs or complementary PWM pairs

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_INDEPENDENT_OPERATION,
           UWord16 param);
```

Arguments:

Table 5-486. PWM_SET_INDEPENDENT_OPERATION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01 Use PWM_ZERO_MASK to set all PWM channels as complementary pairs.

Description: The *PWM_SET_INDEPENDENT_OPERATION* ioctl command determines whether the motor control PWM channels are independent PWMs or complementary PWM pairs. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). The specified channels are set as independent PWMs, the remaining ones are set as complementary PWM pairs. This command manipulates the INDEP bits (Bit 3-1) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.

Design/Implementation: The *PWM_SET_INDEPENDENT_OPERATION* ioctl command is implemented as a macro.

Example 5-430. PWM_SET_INDEPENDENT_OPERATION

```
ioctl(PWM_A, PWM_SET_INDEPENDENT_OPERATION, PWM_CHANNEL_45);
```

This code sets PWM A channel four to five as independent PWMs.

```
ioctl(PWM_B, PWM_SET_INDEPENDENT_OPERATION, PWM_ZERO_MASK);
```

This code sets all PWM B channels as complementary PWM pairs.

5.14.3.30 PWM_SET_INDEPENDENT_MODE - set the selected PWM channel(s) as independent PWMs

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_INDEPENDENT_MODE,
           UWord16 param);
```

Arguments:

Table 5-487. PWM_SET_INDEPENDENT_MODE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired PWM channels operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01.

Description: The *PWM_SET_INDEPENDENT_MODE* ioctl command sets the specified channels as independent PWMs. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). This command sets the appropriate INDEP bits (Bit 3-1) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_INDEPENDENT_MODE* ioctl command is implemented as a macro.

Example 5-431. PWM_SET_INDEPENDENT_MODE

```
ioctl(PWM_A, PWM_SET_INDEPENDENT_MODE, PWM_CHANNEL_45);
```

This code sets PWM A channel four to five as independent PWMs.

```
ioctl(PWM_B, PWM_SET_INDEPENDENT_MODE,
           PWM_CHANNEL_01 | PWM_CHANNEL_23 | PWM_CHANNEL_45);
```

This code sets all PWM B channels as independent PWMs.

5.14.3.31 PWM_SET_COMPLEMENTARY_MODE - set the selected PWM channel(s) as complementary PWM pairs

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_COMPLEMENTARY_MODE,
           UWord16 param);
```

Arguments:

Table 5-488. PWM_SET_COMPLEMENTARY_MODE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired PWM channels operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01.

Description: The *PWM_SET_COMPLEMENTARY_MODE* ioctl command sets the specified channels as complementary PWM pairs. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). This command clears the appropriate INDEP bits (Bit 3-1) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_COMPLEMENTARY_MODE* ioctl command is implemented as a macro.

Example 5-432. PWM_SET_COMPLEMENTARY_MODE

```
ioctl(PWM_A, PWM_SET_COMPLEMENTARY_MODE, PWM_CHANNEL_01);
```

This code sets PWM A channel zero to one as complementary PWM pair.

```
ioctl(PWM_B, PWM_SET_COMPLEMENTARY_MODE,
           PWM_CHANNEL_01 | PWM_CHANNEL_23 | PWM_CHANNEL_45);
```

This code sets all PWM B channels as complementary PWM pairs.

5.14.3.32 PWM_SET_WRITE_PROTECT - write-protect the relevant PWM registers

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_WRITE_PROTECT, NULL);
```

Arguments:

Table 5-489. PWM_SET_WRITE_PROTECT ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_SET_WRITE_PROTECT* ioctl command enables write protection to be used for all write-protectable PWM registers. The write-protected PWM registers are: PWM Disable Mapping Register 1 and 2, PWM Deadtime Register(s), PWM Configure Register and PWM Channel Control Register. This command sets the WP bit (Bit 0) in the PWM Configure Register.

Returns: None.

Range Issues: None.

Special Issues: The WP bit can be cleared only by a reset.

Design/Implementation: The *PWM_SET_WRITE_PROTECT* ioctl command is implemented as a macro.

Example 5-433. PWM_SET_WRITE_PROTECT

```
ioctl(PWM_A, PWM_SET_WRITE_PROTECT, NULL);
```

This code write-protects corresponding PWM A registers.

```
ioctl(PWM_B, PWM_SET_WRITE_PROTECT, NULL);
```

This code write-protects corresponding PWM B registers.

5.14.3.33 PWM_HARDWARE_ACCELERATION - enable or disable the hardware accelerator feature

Call(s):

```
void ioctl(const int *pModuleBase, PWM_HARDWARE_ACCELERATION,
           UWord16 param);
```

Arguments:

Table 5-490. PWM_HARDWARE_ACCELERATION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use PWM_ENABLE to enable the PWM hardware acceleration feature or PWM_DISABLE to disable it.

Description: The *PWM_HARDWARE_ACCELERATION* ioctl command enables or disables the PWM hardware acceleration feature, i.e. setting of the value load mode and swapping. This command sets the ENHA bit (Bit 15) in the PWM Channel Control Register when PWM_ENABLE is used as a parameter. This command clears the above mentioned bit in case of PWM_DISABLE.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_HARDWARE_ACCELERATION* ioctl command is implemented as a macro.

Example 5-434. PWM_HARDWARE_ACCELERATION

```
ioctl(PWM_A, PWM_HARDWARE_ACCELERATION, PWM_ENABLE);
```

This code enables PWM A generator and the corresponding pins.

```
ioctl(PWM_B, PWM_HARDWARE_ACCELERATION, PWM_DISABLE);
```

This code disables PWM B generator and the corresponding pins.

5.14.3.34 PWM_SET_CHANNEL_MASK - mask PWM logical channels

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_CHANNEL_MASK,
           pwm_tPWMSignalMask param);
```

Arguments:

Table 5-491. PWM_SET_CHANNEL_MASK ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the PWM signal-logical channel. Parameter is of *pwm_tPWMSignalMask* type and contains six control bits representing six PWM signals.

Description: The *PWM_SET_CHANNEL_MASK* ioctl command determines the mask for each of the PWM logical channels. The PWM signals-logical channels specified by *param* are masked, i.e. are set to a value of 0% duty cycle. The remaining PWM logical channels are unmasked. This command sets the MSK bits (Bit 13-8) in the PWM Channel Control Register.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.

Design/Implementation: The *PWM_SET_CHANNEL_MASK* ioctl command is implemented as a macro.

Example 5-435. PWM_SET_CHANNEL_MASK

```
pwm_tPWMSignalMask          mask;

mask = MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_3;

ioctl(PWM_A, PWM_SET_CHANNEL_MASK, mask);
```

This code masks PWM2 and PWM3 logical channels of the PWM A module.

```
ioctl(PWM_B, PWM_SET_CHANNEL_MASK,
           MC_PWM_SIGNAL_2 | MC_PWM_SIGNAL_5);
```

This code masks PWM2 and PWM5 logical channels of the PWM B module.

```
ioctl(PWM_B, PWM_SET_CHANNEL_MASK, MC_PWM_NO_SIGNALS);
```

This code unmasks all PWM logical channels of the PWM B module.

5.14.3.35 PWM_SET_LOAD_MODE - set PWM Value Registers load mode

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_LOAD_MODE,
           UWord16 param);
```

Arguments:

Table 5-492. PWM_SET_LOAD_MODE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use one of the predefined constants: PWM_LOAD_INDEP / PWM_LOAD_FROM_0_TO_5 / PWM_LOAD_FROM_0_TO_3

Description: The *PWM_SET_LOAD_MODE* ioctl command determines the way by which the PWM Value Registers are loaded. This command manipulates the VLMODE bits (Bit 5-4) in the PWM Channel Control Register.

Returns: None.

Range Issues: None.

Special Issues: Hardware acceleration must be enabled (set the ENHA bit in the PWM Channel Control Register, e.g. using the PWM_HARDWARE_ACCELERATION command) prior to this command.

Design/Implementation: The *PWM_SET_LOAD_MODE* ioctl command is implemented as a macro.

Example 5-436. PWM_SET_LOAD_MODE

```
ioctl(PWM_A, PWM_SET_LOAD_MODE, PWM_LOAD_INDEP);
```

This code specifies that each PWM A Value Register must be loaded independently.

```
ioctl(PWM_B, PWM_SET_LOAD_MODE, PWM_LOAD_FROM_0_TO_3);
```

This code specifies that when PWM B Value Register 0 is loaded, this value is also copied to the PWM Value Registers 1 to 3 (these four PWM Value Registers contain the same value).

5.14.3.36 PWM_SET_SWAP - swap the selected PWM channels

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_SWAP, UWord16 param);
```

Arguments:

Table 5-493. PWM_SET_SWAP ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use combination of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01 Use PWM_ZERO_MASK to set no swapping between PWM channels.

Description: The *PWM_SET_SWAP* ioctl command swaps the selected PWM channels. The remaining PWM channels are not swapped. Each pair of PWM channels can be configured: channel zero to one (PWM_CHANNEL_01), channel two to three (PWM_CHANNEL_23) and channel four to five (PWM_CHANNEL_45). This command manipulates the SWP bits (Bit 2-0) in the PWM Channel Control Register.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.
Hardware acceleration must be enabled (set the ENHA bit in the PWM Channel Control Register, e.g. using the PWM_HARDWARE_ACCELERATION command) prior to this command.

Design/Implementation: The *PWM_SET_SWAP* ioctl command is implemented as a macro.

Example 5-437. PWM_SET_SWAP

```
ioctl(PWM_A, PWM_SET_SWAP, PWM_CHANNEL_45);
```

This code swaps PWM A channel four and PWM A channel five.

```
ioctl(PWM_B, PWM_SET_SWAP, PWM_ZERO_MASK);
```

This code sets no swap between PWM B channels.

5.14.3.37 PWM_WRITE_VALUE_REG_x - write new value to PWM Value Register x

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WRITE_VALUE_REG_x,
           Word16 param);
```

Arguments:

Table 5-494. PWM_WRITE_VALUE_REG_x ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value representing the PWM pulse width in PWM clock period.

Description: The *PWM_WRITE_VALUE_REG_x* ioctl command writes a new value, representing the PWM pulse width in PWM clock periods, to the PWM Channel x Value Register (PWMVALx). Note: x represents the Value Register number corresponding to the PWM channel (0 - 5) and it is part of the command name.

Returns: None.

Range Issues: A value (*param*) less than or equal to zero deactivates the PWM output for the entire PWM period.

Special Issues: The written value does not take effect until the LDOK bit is set and the next PWM load cycle begins.

Design/Implementation: The *PWM_WRITE_VALUE_REG_x* ioctl command is implemented as a macro.

Example 5-438. PWM_WRITE_VALUE_REG_x

```
Word16          pulse;

pulse = 0x4000;

ioctl(PWM_A, PWM_WRITE_VALUE_REG_0, pulse);
```

This code writes a new value to the PWM A Value Register 0.

```
ioctl(PWM_B, PWM_WRITE_VALUE_REG_4, 0x7fff);
```

This code writes 0x7fff to the PWM B Value Register 4.

5.14.3.38 PWM_WRITE_VALUE_REGS_COMPL - write new values to the PWM Value Registers 0, 2 and 4

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WRITE_VALUE_REGS_COMPL,
           pwm_sComplementaryValues *pParam);
```

Arguments:

Table 5-495. PWM_WRITE_VALUE_REGS_COMPL ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to the structure of *pwm_sComplementaryValues* type containing the new PWM pulse widths in PWM clock period for the PWM Value Registers 0, 2 and 4.

Description: The *PWM_WRITE_VALUE_REGS_COMPL* ioctl command writes the new values, representing the PWM pulse widths expressed in PWM clock periods, to the PWM Channel 0, 2 and 4 Value Registers (PWMVAL0, PWMVAL2 and PWMVAL4). This command should be used in PWM complementary pair operation.

Returns: None.

Range Issues: A value less than or equal to zero deactivates the PWM output for the entire PWM period.

Special Issues: The written values do not take effect until the LDOK bit is set and the next PWM load cycle begins.

Design/Implementation: The *PWM_WRITE_VALUE_REGS_COMPL* ioctl command is implemented as a macro.

Example 5-439. PWM_WRITE_VALUE_REGS_COMPL

```
pwm_sComplementaryValues compVal;

compVal.pwmChannel_0_Value = 0x2000;
compVal.pwmChannel_2_Value = 0x4000;
compVal.pwmChannel_4_Value = 0x6000;

ioctl(PWM_A, PWM_WRITE_VALUE_REGS_COMPL, &compVal);
```

This code writes the new values (pulse widths) to the PWM A Value Registers 0, 2 and 4.

5.14.3.39 PWM_WRITE_VALUE_REGS_INDEP - write new values to the PWM Value Registers 0, 1, 2, 3, 4 and 5

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WRITE_VALUE_REGS_INDEP,
           pwm_sIndependentValues *pParam);
```

Arguments:

Table 5-496. PWM_WRITE_VALUE_REGS_INDEP ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to the structure of *pwm_sIndependentValues* type containing the new PWM pulse widths in PWM clock period for the PWM Value Registers 0, 1, 2, 3, 4 and 5.

Description: The *PWM_WRITE_VALUE_REGS_INDEP* ioctl command writes the new values representing the PWM pulse widths expressed in PWM clock periods, to the PWM Channel 0, 1, 2, 3, 4 and 5 Value Registers (PWMVAL0, PWMVAL1, PWMVAL2, PWMVAL3, PWMVAL4 and PWMVAL5). This command should be used in independent PWMs operation.

Returns: None.

Range Issues: A value less than or equal to zero deactivates the PWM output for the entire PWM period.

Special Issues: The written values do not take effect until the LDOK bit is set and the next PWM load cycle begins.

Design/Implementation: The *PWM_WRITE_VALUE_REGS_INDEP* ioctl command is implemented as a macro.

Example 5-440. PWM_WRITE_VALUE_REGS_INDEP

```
pwm_sIndependentValues indepVal;

indepVal.pwmChannel_0_Value = 0x1000;
indepVal.pwmChannel_1_Value = 0x2000;
indepVal.pwmChannel_2_Value = 0x3000;
indepVal.pwmChannel_3_Value = 0x4000;
indepVal.pwmChannel_4_Value = 0x5000;
indepVal.pwmChannel_5_Value = 0x6000;

ioctl(PWM_A, PWM_WRITE_VALUE_REGS_INDEP, &indepVal);
```

This code writes the new values (pulse widths) to the PWM A Value Registers 0, 1, 2, 3, 4 and 5.

5.14.3.40 PWM_READ_FAULT_STATUS_REG - return the content of the PWM Fault Status & Acknowledge Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_READ_FAULT_STATUS_REG,
          NULL);
```

Arguments:

Table 5-497. PWM_READ_FAULT_STATUS_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_READ_FAULT_STATUS_REG* ioctl command returns the content of the PWM Fault Status & Acknowledge Register (PMFSA).

Returns: the content of the PWM Fault Status & Acknowledge Register as UWord16.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_READ_FAULT_STATUS_REG* ioctl command is implemented as a macro.

Example 5-441. PWM_READ_FAULT_STATUS_REG

```
UWord16 statusWord;

statusWord = ioctl(PWM_A, PWM_READ_FAULT_STATUS_REG, NULL);
```

This code stores the content of the PWM A Fault Status & Acknowledge Register to the variable statusWord.

5.14.3.41 PWM_READ_COUNTER_REG - return the content of the PWM Counter Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_READ_COUNTER_REG,
          NULL);
```

Arguments:

Table 5-498. PWM_READ_COUNTER_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_READ_COUNTER_REG* ioctl command returns the content of the PWM Counter Register (PMCNT).

Returns: he content of the PWM Counter Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_READ_COUNTER_REG* ioctl command is implemented as a macro.

Example 5-442. PWM_READ_COUNTER_REG

```
UWord16 counter;

counter = ioctl(PWM_A, PWM_READ_COUNTER_REG, NULL);
```

This code stores the content of the PWM A Counter Register to the variable counter.

5.14.3.42 PWM_READ_CONTROL_REG - return the content of the PWM Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_READ_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-499. PWM_READ_CONTROL_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_READ_CONTROL_REG* ioctl command returns the content of the PWM Control Register (PMCTL).

Returns: the content of the PWM Control Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-443. PWM_READ_CONTROL_REG

```
UWord16 controlWord;

controlWord = ioctl(PWM_A, PWM_READ_CONTROL_REG, NULL);
```

This code stores the content of the PWM A Control Register to the variable controlWord.

5.14.3.43 PWM_READ_PORT_REG - return the content of the PWM Port Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_READ_PORT_REG,
              NULL);
```

Arguments:

Table 5-500. PWM_READ_PORT_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_READ_PORT_REG* ioctl command returns the content of the PWM Port Register (PMPORT).

Returns: the content of the PWM Port Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_READ_PORT_REG* ioctl command is implemented as a macro.

Example 5-444. PWM_READ_PORT_REG

```
UWord16 portWord;

portWord = ioctl(PWM_A, PWM_READ_COUNTER_REG, NULL);
```

This code stores the content of the PWM A Port Register to the variable portWord.

5.14.3.44 PWM_GET_CURRENT_STATUS_INPUTS - return values of the three current status inputs

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
                PWM_GET_CURRENT_STATUS_INPUTS, NULL);
```

Arguments:

Table 5-501. PWM_GET_CURRENT_STATUS_INPUTS ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_GET_CURRENT_STATUS_INPUTS* ioctl command returns values of the three current status inputs - ISx bits (Bit 6, 5 and 4) in the PWM Port Register (PMPORT).

Returns: the values of the three current status inputs in their original bit position in the PWM Port Register as UWord16.

Range Issues: None.

Special Issues: This command is applicable only if the chip has IS pins.

Design/Implementation: The *PWM_GET_CURRENT_STATUS_INPUTS* ioctl command is implemented as a macro.

Example 5-445. PWM_GET_CURRENT_STATUS_INPUTS

```
UWord16 isStatus;

isStatus = ioctl(PWM_A, PWM_GET_CURRENT_STATUS_INPUTS, NULL);
```

This code stores the status of the ISx bits in the PWM A Port Register to the variable isStatus.

5.14.3.45 PWM_GET_FAULT_INPUTS - return values of the four fault inputs

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_GET_FAULT_INPUTS,
      NULL);
```

Arguments:

Table 5-502. PWM_GET_FAULT_INPUTS ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_GET_FAULT_INPUTS* ioctl command returns values of the four fault inputs - FAULTx bits (Bit 3, 2, 1 and 0) in the PWM Port Register (PMPORT).

Returns: the values of the four fault inputs in their original bit position in the PWM Port Register as UWord16.

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_GET_FAULT_INPUTS* ioctl command is implemented as a macro.

Example 5-446. PWM_GET_FAULT_INPUTS

```
UWord16 faultStatus;

faultStatus = ioctl(PWM_A, PWM_GET_FAULT_INPUTS, NULL);
```

This code stores the status of the FAULTx bits in the PWM A Port Register to the variable faultStatus.

5.14.3.46 PWM_GET_FAULT_INPUT_x - return value of the FAULTx input

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PWM_GET_FAULT_INPUT_x,
              NULL);
```

Arguments:

Table 5-503. PWM_GET_FAULT_INPUT_x ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_GET_FAULT_INPUT_x* ioctl command returns value of the corresponding fault input - FAULTx bit (Bit 3, 2, 1 or 0) in the PWM Port Register (PMPORT).

Returns: the value of the fault input in its original bit position in the PWM Port Register as UWord16 (i.e. true if set, false otherwise).

Range Issues: None.

Special Issues: Note that not all FAULT pins are available on all chips.

Design/Implementation: The *PWM_GET_FAULT_INPUT_x* ioctl command is implemented as a macro.

Example 5-447. PWM_GET_FAULT_INPUT_x

```
UWord16 faultStatus2;

faultStatus2 = ioctl(PWM_A, PWM_GET_FAULT_INPUT_2, NULL);
```

This code stores the status of the FAULT2 bit in the PWM A Port Register to the variable faultStatus2.

5.14.3.47 PWM_SOFTWARE_OUTPUTS_CONTROL - set software control of selected PWM output pins and activates/deactivates them

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SOFTWARE_OUTPUTS_CONTROL,
           pwm_sOutputControl *pParam);
```

Arguments:

Table 5-504. PWM_SOFTWARE_OUTPUTS_CONTROL ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to structure of *pwm_sOutputControl* type containing the mask (*SoftwareControlled*) to enable software control of corresponding PWM pins and the mask (*OutputControl*) to control the PWM output pins.

Description: The *PWM_SOFTWARE_OUTPUTS_CONTROL* ioctl command enables software control of selected PWM pins and controls their state. This command sets the PAD_EN bit (Bit 15), manipulates the OUTCTL bits (Bit 13-8) and the OUT bits (Bit 5-0) in the PWM Output Control Register. For more information see also Section 5.14.3.17 and Section 5.14.3.18.

Returns: None.

Range Issues: None.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.

Design/Implementation: The *PWM_SOFTWARE_OUTPUTS_CONTROL* ioctl command is implemented as a macro.

Example 5-448. PWM_SOFTWARE_OUTPUTS_CONTROL

```
pwm_sOutputControl outCtrl;

outCtrl.SoftwareControlled = MC_PWM_SIGNAL_0 | MC_PWM_SIGNAL_1;
outCtrl.OutputControl = MC_PWM_SIGNAL_0 | MC_PWM_SIGNAL_1;

ioctl(PWM_A, PWM_SOFTWARE_OUTPUTS_CONTROL, &outCtrl);
```

This code enables software control of PWM A pins PWM0 and PWM1 and activates both pins in the independent PWMs operation or activates the PWM0 pin while PWM1 is the complement to PWM0 in the complementary PWM pairs mode.

5.14.3.48 PWM_UPDATE_VALUE_REG_x - update the PWM Value Register x by new value in percentage

Call(s):

```
void ioctl(const int *pModuleBase, PWM_UPDATE_VALUE_REG_x,
           Word16 param);
```

Arguments:

Table 5-505. PWM_UPDATE_VALUE_REG_x ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Value representing the PWM duty cycle in percentage.

Description: The *PWM_UPDATE_VALUE_REG_x* ioctl command recalculates the input value (*param*) of the duty cycle, in percentage, with respect to the PWM modulus (see the expression below) and writes the resulting value to the PWM Channel x Value Register (PWMVALx). The LDOK bit (Bit 1 in the PWM Control Register) is set afterwards, so this action takes effect in the next PWM reload cycle. Note: x represents the Value Register number corresponding to the PWM channel (0 - 5) and it is part of the command name.

The actual value of the PWM modulus is read from the PWM Counter Modulo Register (PWMCM).

$$\text{PWM Value} = \text{PWM Duty Cycle} \times \text{PWM Modulus} / 2^{15}$$

Returns: None.

Range Issues: The data value representing the duty cycle in percentage is scaled into the effective data range, where 0x0000 corresponds to 0% and 0x7fff corresponds to 100%. A value less than or equal to zero deactivates the PWM output for the entire PWM period.

Special Issues: None.

Design/Implementation: The *PWM_UPDATE_VALUE_REG_x* ioctl command is implemented as a macro.

Example 5-449. PWM_UPDATE_VALUE_REG_x

```
Word16 pulse;
pulse = 0x4000; /* 50% */

ioctl(PWM_A, PWM_UPDATE_VALUE_REG_0, pulse);
```

This code writes a new value to the PWM A Value Register 0 which actually represents 50% of duty cycle and sets the LDOK bit in the PWM Control Register.

```
ioctl(PWM_B, PWM_UPDATE_VALUE_REG_3, 0x2000); /* 25% */
```

This code writes 0x2000 to the PWM B Value Register 3, which actually represents 25% of duty cycle and sets the LDOK bit in the PWM Control Register.

5.14.3.49 PWM_UPDATE_VALUE_REGS_COMPL - update the PWM Value Registers 0, 2 and 4 by the new values in percentage

Call(s):

```
void ioctl(const int *pModuleBase, PWM_UPDATE_VALUE_REGS_COMPL,
           pwm_sComplementaryValues *pParam);
```

Arguments:

Table 5-506. PWM_UPDATE_VALUE_REGS_COMPL ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to structure of *pwm_sComplementaryValues* type containing the new values representing the PWM duty cycles in percentage to update the PWM Value Registers 0, 2 and 4.

Description: The *PWM_UPDATE_VALUE_REGS_COMPL* ioctl command recalculates the input values of duty cycles, in percentage, with respect to the PWM modulus (see the expression below) and writes the resulting values to the three corresponding PWM Value Registers - PWM Channel 0,2,4 Value Registers. The LDOK bit (Bit 1 in PWM Control Register) is set afterwards, so this action takes effect in the next PWM reload cycle. This command should be used in PWM complementary pairs operation.

The actual value of the PWM modulus is read from the PWM Counter Modulo Register (PWMCM).

$$\text{PWM Value} = \text{PWM Duty Cycle} \times \text{PWM Modulus} / 2^{15}$$

Returns: None.

Range Issues: The data values representing the duty cycles in percentage are scaled into the effective data range, where 0x0000 corresponds to 0% and 0x7fff corresponds to 100%. Values less than or equal to zero deactivate the PWM output for the entire PWM period.

Special Issues: None.

Design/Implementation: The *PWM_UPDATE_VALUE_REGS_COMPL* ioctl command is implemented as a macro.

Example 5-450. PWM_UPDATE_VALUE_REGS_COMPL

```
pwm_sComplementaryValues compVal;

compVal.pwmChannel_0_Value = 0x2000; /* 25% */
compVal.pwmChannel_2_Value = 0x4000; /* 50% */
compVal.pwmChannel_4_Value = 0x6000; /* 75% */

ioctl(PWM_A, PWM_UPDATE_VALUE_REGS_COMPL, &compVal);
```

This code writes the new values to the PWM A Value Registers 0, 2 and 4, which actually represent the desired duty cycles in percentage and sets the LDOK bit in the PWM Control Register.

5.14.3.50 PWM_UPDATE_VALUE_REGS_INDEP - update the PWM Value Registers 0, 1, 2, 3, 4 and 5 by the new values in percentage

Call(s):

```
void ioctl(const int *pModuleBase, PWM_UPDATE_VALUE_REGS_INDEP,
           pwm_sIndependentValues *pParam);
```

Arguments:

Table 5-507. PWM_UPDATE_VALUE_REGS_INDEP ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to structure of *pwm_sIndependentValues* type containing the new values representing the PWM duty cycles in percentage to update the PWM Value Registers 0, 1, 2, 3, 4 and 5.

Description: The *PWM_UPDATE_VALUE_REGS_INDEP* ioctl command recalculates the input values of duty cycles, in percentage, with respect to the PWM modulus (see the expression below) and writes the resulting values to the six corresponding PWM Value Registers - PWM Channel 0,1,2,3,4,5 Value Registers. The LDOK bit (Bit 1 in PWM Control Register) is set afterwards, so this action takes effect in the next PWM reload cycle. This command should be used in independent PWMs operation.

The actual value of the PWM modulus is read from the PWM Counter Modulo Register (PWMCM).

$$\text{PWM Value} = \text{PWM Duty Cycle} \times \text{PWM Modulus} / 2^{15}$$

Returns: None.

Range Issues: The data values representing the duty cycles in percentage are scaled into the effective data range, where 0x0000 corresponds to 0% and 0x7fff corresponds to 100%. Values less than or equal to zero deactivate the PWM output for the entire PWM period.

Special Issues: None.

Design/Implementation: The *PWM_UPDATE_VALUE_REGS_INDEP* ioctl command is implemented as a macro.

Example 5-451. PWM_UPDATE_VALUE_REGS_INDEP

```
pwm_sIndependentValues indepVal;
indepVal.pwmChannel_0_Value = 0x0000; /* 0% */
indepVal.pwmChannel_1_Value = 0x2000; /* 25% */
indepVal.pwmChannel_2_Value = 0x4000; /* 50% */
indepVal.pwmChannel_3_Value = 0x7fff; /* 100% */
indepVal.pwmChannel_4_Value = 0x6000; /* 75% */
indepVal.pwmChannel_5_Value = 0x2000; /* 25% */
ioctl(PWM_A, PWM_UPDATE_VALUE_REGS_INDEP, &indepVal);
```

This code writes new values to the PWM A Value Registers 0, 1, 2, 3, 4 and 5, which actually represent the desired duty cycles in percentage and sets the LDOK bit in the PWM Control Register.

5.14.3.51 PWM_UPDATE_VALUE_SET_VLMODE - update the PWM Value Register 0 by new value in percentage and set the PWM Value Registers Load Mode

Call(s):

```
void ioctl(const int *pModuleBase, PWM_UPDATE_VALUE_SET_VLMODE,
           pwm_sUpdateValueSetVlmode *pParam);
```

Arguments:

Table 5-508. PWM_UPDATE_VALUE_SET_VLMODE ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to the structure of *pwm_sUpdateValueSetVlmode* type containing the new value of PWM duty cycle in percentage (*DutyCycle*) and the value representing the Value Register Load Mode (*Vlmode*).

Description: The *PWM_UPDATE_VALUE_SET_VLMODE* ioctl command recalculates the value of the duty cycle, in percentage, with respect to the PWM modulus (see the expression below) and writes the resulting value to the PWM Channel 0 Value Register (PWMVAL0). The value of the Value Register Load Mode determines the way the PWM Value Registers are being loaded (see also Section 5.14.3.35) in the PWM Channel Control Register. The LDOK bit (Bit 1 in PWM Control Register) is set afterwards, so this action takes effect in the next PWM reload cycle.

The actual value of the PWM modulus is read from the PWM Counter Modulo Register (PWMCM).

$$\text{PWM Value} = \text{PWM Duty Cycle} \times \text{PWM Modulus} / 2^{15}$$

Returns: None.

Range Issues: The data value representing the duty cycle in percentage is scaled into the effective data range, where 0x0000 corresponds to 0% and 0x7fff corresponds to 100%. A value less than or equal to zero deactivates the PWM output for the entire PWM period.

Special Issues: Ensure that this *ioctl* call cannot be interrupted.
Hardware acceleration must be enabled (set the ENHA bit in the PWM Channel Control Register, e.g. using the PWM_HARDWARE_ACCELERATION command) prior to this command.

Design/Implementation: The *PWM_UPDATE_VALUE_SET_VLMODE* ioctl command is implemented as a macro.

Example 5-452. PWM_UPDATE_VALUE_SET_VLMODE

```
pwm_sUpdateValueSetVlmode  data;

data.DutyCycle = 0x2000; /* 25% */
data.Vlmode = PWM_LOAD_FROM_0_TO_3;

ioctl(PWM_A, PWM_UPDATE_VALUE_SET_VLMODE, &data);
```

This code writes a new value to the PWM A Value Register 0, which actually represents 25% of duty cycle. This value is also copied to PWM A Value Registers 1, 2 and 3, and sets the LDOK bit in the PWM Control Register.

5.14.3.52 PWM_SET_MASK_SWAP - mask the selected PWM logical channels and set the swapping channel operation

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_MASK_SWAP,
           pwm_sChannelControl *pParam);
```

Arguments:

Table 5-509. PWM_SET_MASK_SWAP ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
*pParam	in	Pointer to structure of *pwm_sChannelControl* type containing the new mask (*Mask*) of the PWM logical channels and the mask (*Swap*) to determine channel swapping operation.

Description: The *PWM_SET_MASK_SWAP* ioctl command determines the mask for each of the PWM logical channels. The PWM signals-logical channels specified by *Mask* are masked, i.e. are set to a value of 0% duty cycle. The remaining PWM logical channels are unmasked. This command also swaps (*Swap*) the selected PWM channels. The remaining PWM channels are not swapped. This command manipulates the MSK bits (Bit 13-8) and the SWP bits (Bit 2-0) in the PWM Channel Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command should be used in the independent mode. In complementary mode the masked channels are set to 0% duty cycle and the corresponding complementary channels are therefore set to 100% duty cycle.
Hardware acceleration must be enabled (set the ENHA bit in the PWM Channel Control Register, e.g. using the PWM_HARDWARE_ACCELERATION command) prior to this command.

Design/Implementation: The *PWM_SET_MASK_SWAP* ioctl command is implemented as a macro.

Example 5-453. PWM_SET_MASK_SWAP

```
pwm_sChannelControl data;

data.Mask = MC_PWM_SIGNAL_0 | MC_PWM_SIGNAL_1;
data.Swap = PWM_CHANNEL_45;

ioctl(PWM_A, PWM_SET_MASK_SWAP, &data);
```

This code masks PWM0 and PWM1 signals of PWM A and simultaneously the PWM4 and PWM5 signals-channels are swapped.

5.14.3.53 PWM_DEBUG_OPERATION - enable or disable PWM operation during debug mode

Call(s):

```
void ioctl(const int *pModuleBase, PWM_DEBUG_OPERATION,
           UWord16 param);
```

Arguments:

Table 5-510. PWM_DEBUG_OPERATION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use PWM_RUN to enable the PWM operation during debug mode or PWM_STOP to disable it.

Description: The *PWM_DEBUG_OPERATION* ioctl command enables or disables the PWM operation during debug mode. This command sets the DBG_EN bit (Bit 14) in the PWM Configure Register (PMCFG) when PWM_RUN is used as a parameter. This command clears the above mentioned bit in case of PWM_STOP.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_DEBUG_OPERATION* ioctl command is implemented as a macro.

Example 5-454. PWM_DEBUG_OPERATION

```
ioctl(PWM_A, PWM_DEBUG_OPERATION, PWM_RUN);
```

This code enables PWM A operation during debug mode.

```
ioctl(PWM_B, PWM_DEBUG_OPERATION, PWM_STOP);
```

This code disables PWM B operation during debug mode.

5.14.3.54 PWM_WAIT_OPERATION - enable or disable PWM operation during WAIT mode

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WAIT_OPERATION,
           UWord16 param);
```

Arguments:

Table 5-511. PWM_WAIT_OPERATION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use PWM_RUN to enable the PWM operation during WAIT mode or PWM_STOP to disable it.

Description: The *PWM_WAIT_OPERATION* ioctl command enables or disables the PWM operation during WAIT mode. This command sets the WAIT_EN bit (Bit 13) in the PWM Configure Register (PMCFG) when PWM_RUN is used as a parameter. This command clears the above mentioned bit in case of PWM_STOP.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_WAIT_OPERATION* ioctl command is implemented as a macro.

Example 5-455. PWM_WAIT_OPERATION

```
ioctl(PWM_A, PWM_WAIT_OPERATION, PWM_RUN);
```

This code enables PWM A operation during WAIT mode.

```
ioctl(PWM_B, PWM_WAIT_OPERATION, PWM_STOP);
```

This code disables PWM B operation during WAIT mode.

5.14.3.55 PWM_MASK_SWAP_OPERATION - enable or disable enhanced SWAP and MASK operations

Call(s):

```
void ioctl(const int *pModuleBase, PWM_MASK_SWAP_OPERATION,
           UWord16 param);
```

Arguments:

Table 5-512. PWM_MASK_SWAP_OPERATION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use PWM_ENHANCED to enable improved SWAP and MASK operations or PWM_80X_COMPATIBLE to disable it.

Description: The *PWM_MASK_SWAP_OPERATION* ioctl command enables or disables the improved SWAP and MASK operations. See the device user's manual for details. This command sets the nBX bit (Bit 14) in the PWM Channel Control Register (PMCCR) when PWM_ENHANCED is used as a parameter. This command clears the above mentioned bit in case of PWM_80X_COMPATIBLE.

Returns: None.

Range Issues: None.

Special Issues: Hardware acceleration must be enabled (set the ENHA bit in the PWM Channel Control Register, e.g. using the PWM_HARDWARE_ACCELERATION command) prior to this command.

Design/Implementation: The *PWM_MASK_SWAP_OPERATION* ioctl command is implemented as a macro.

Example 5-456. PWM_MASK_SWAP_OPERATION

```
ioctl(PWM_A, PWM_MASK_SWAP_OPERATION, PWM_ENHANCED);
```

This code enables improved SWAP and MASK operations on the PWM A module.

```
ioctl(PWM_B, PWM_MASK_SWAP_OPERATION, PWM_80X_COMPATIBLE);
```

This code disables improved SWAP and MASK operations on the PWM B module.

5.14.3.56 PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION - set half cycle internal correction method

Call(s):

```
void ioctl(const int *pModuleBase,
     PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION, UWord16 param);
```

Arguments:

Table 5-513. PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired PWM pair. Use one of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01

Description: The *PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION* ioctl command sets the half cycle internal correction method for the selected PWM pair(s). See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details. This command sets the ICCx bits (Bit 2, 1, 0) in the PWM Internal Correction Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION* ioctl command is implemented as a macro.

Example 5-457. PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION

```
ioctl(PWM_A, PWM_SET_HALF_CYCLE_INTERNAL_CORRECTION,
                PWM_CHANNEL_01 | PWM_CHANNEL_23);
```

This code sets half cycle internal correction method for PWM pairs 0-1 and 2-3 on PWM A.

5.14.3.57 PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION - set full cycle internal correction method

Call(s):

```
void ioctl(const int *pModuleBase,
     PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION, UWord16 param);
```

Arguments:

Table 5-514. PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired PWM pair. Use one of the predefined constants: PWM_CHANNEL_45 | PWM_CHANNEL_23 | PWM_CHANNEL_01

Description: The *PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION* ioctl command sets the full cycle internal correction method for the selected PWM pair(s). See the **MC56F8300 Peripheral User Manual** or the **56F8000 Peripheral Reference Manual** for more details. This command clears the ICCx bits (Bit 2, 1, 0) in the PWM Internal Correction Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION* ioctl command is implemented as a macro.

Example 5-458. PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION

```
ioctl(PWM_A, PWM_SET_FULL_CYCLE_INTERNAL_CORRECTION,
                    PWM_CHANNEL_45);
```

This code sets full cycle internal correction method for PWM pair 4-5 on PWM A.

5.14.3.58 PWM_READ_INTERNAL_CORRECTION_CONTROL_REG - return the content of the PWM Internal Correction Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
              PWM_READ_INTERNAL_CORRECTION_CONTROL_REG, NULL);
```

Arguments:

Table 5-515. PWM_READ_INTERNAL_CORRECTION_CONTROL_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.

Description: The *PWM_READ_INTERNAL_CORRECTION_CONTROL_REG* ioctl command returns the content of the PWM Internal Correction Control Register (PMICCR).

Returns: the content of the PWM Internal Correction Control Register as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *PWM_READ_INTERNAL_CORRECTION_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-459. PWM_READ_INTERNAL_CORRECTION_CONTROL_REG

```
UWord16 icontrolWord;

icontrolWord = ioctl(PWM_A,
              PWM_READ_INTERNAL_CORRECTION_CONTROL_REG, NULL);
```

This code stores the content of the PWM A Internal Correction Control Register to the variable icontrolWord.

5.14.3.59 PWM_SET_SOURCE_0 - set the source signal for the complementary PWM0/PWM1 pair output

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_SOURCE_0,
           Word16 param);
```

Arguments:

Table 5-516. PWM_SET_SOURCE_0 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use one of the predefined constants: PWM_SOURCE_PWM / PWM_SOURCE_ADC_0 / PWM_SOURCE_PWMSRC_0 / PWM_SOURCE_TMR_0

Description: The *PWM_SET_SOURCE_0* ioctl command sets the source signal for the complementary PWM0/PWM1 pair output. See the **56F8000 Peripheral Reference Manual** for more details. This command modifies the SRC0 bit group (Bit 0-1) in the PWM Source Control Register (SCTRL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx. The PWM Source Control Register can only receive writing when the WP bit in the PWM Configure Register is clear. Make sure that OUTCTL0 and OUTCTL1 (bits 8 and 9 of the PWM Output Control Register) are set when using this command.

Design/Implementation: The *PWM_SET_SOURCE_0* ioctl command is implemented as a macro.

Example 5-460. PWM_SET_SOURCE_0

```
ioctl(PWM, PWM_SET_SOURCE_0, PWM_SOURCE_ADC_0);
```

This code determines that ADC SAMPLE0 result is used as PWM source.

5.14.3.60 PWM_SET_SOURCE_1 - set the source signal for the complementary PWM2/PWM3 pair output

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_SOURCE_1,
           Word16 param);
```

Arguments:

Table 5-517. PWM_SET_SOURCE_1 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use one of the predefined constants: PWM_SOURCE_PWM / PWM_SOURCE_ADC_1 / PWM_SOURCE_PWMSRC_1 / PWM_SOURCE_TMR_1 / PWM_SOURCE_SRC_0

Description: The *PWM_SET_SOURCE_1* ioctl command sets the source signal for the complementary PWM2/PWM3 pair output. See the **56F8000 Peripheral Reference Manual** for more details. This command modifies the SRC1 bit group (Bit 2-4) in the PWM Source Control Register (SCTRL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx. The PWM Source Control Register can only receive writing when the WP bit in the PWM Configure Register is clear. Make sure that OUTCTL2 and OUTCTL3 (bits 10 and 11 of the PWM Output Control Register) are set when using this command.

Design/Implementation: The *PWM_SET_SOURCE_1* ioctl command is implemented as a macro.

Example 5-461. PWM_SET_SOURCE_1

```
ioctl(PWM, PWM_SET_SOURCE_1, PWM_SOURCE_ADC_1);
```

This code determines that ADC SAMPLE1 result is used as PWM source.

5.14.3.61 PWM_SET_SOURCE_2 - set the source signal for the complementary PWM4/PWM5 pair output

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_SOURCE_2,
           Word16 param);
```

Arguments:

Table 5-518. PWM_SET_SOURCE_2 ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use one of the predefined constants: PWM_SOURCE_PWM / PWM_SOURCE_ADC_2 / PWM_SOURCE_PWMSRC_2 / PWM_SOURCE_TMR_2 / PWM_SOURCE_SRC_0

Description: The *PWM_SET_SOURCE_2* ioctl command sets the source signal for the complementary PWM4/PWM5 pair output. See the **56F8000 Peripheral Reference Manual** for more details. This command modifies the SRC2 bit group (Bit 5-7) in the PWM Source Control Register (SCTRL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx. The PWM Source Control Register can only receive writing when the WP bit in the PWM Configure Register is clear. Make sure that OUTCTL4 and OUTCTL5 (bits 12 and 13 of the PWM Output Control Register) are set when using this command.

Design/Implementation: The *PWM_SET_SOURCE_2* ioctl command is implemented as a macro.

Example 5-462. PWM_SET_SOURCE_2

```
ioctl(PWM, PWM_SET_SOURCE_2, PWM_SOURCE_ADC_2);
```

This code determines that ADC SAMPLE2 result is used as PWM source.

5.14.3.62 PWM_SET_COMPARE_INVERT_x - set the polarity of the PWM compare output x

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_COMPARE_INVERT_x,
           Word16 param);
```

Arguments:

Table 5-519. PWM_SET_COMPARE_INVERT_x ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx or PWM_A and PWM_B on MC56F83xx. Note that PWM_B is not available on certain chips.
param	in	Parameter to select the desired operation. Use one of the predefined constants: PWM_NORMAL / PWM_INVERT

Description: The *PWM_SET_COMPARE_INVERT_x* ioctl command controls the polarity of the PWM compare output x. Note: x represents the Value Register number corresponding to the PWM channel (0 - 5) and it is part of the command name. When the PWM_NORMAL value is used as a parameter, the PWM output *x* is driven high when the PWM counter is less then the PWM Value Register *x*. With the PWM_INVERT parameter value, the PWM output is driven high when counter is greater than Value Register.

This command modifies corresponding CINVx bit in the PWM Source Control Register (SCTRL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx. The PWM Source Control Register can only receive writing when the WP bit in the PWM Configure Register is clear.

Design/Implementation: The *PWM_SET_COMPARE_INVERT_x* ioctl command is implemented as a macro.

Example 5-463. PWM_SET_COMPARE_INVERT_x

```
ioctl(PWM, PWM_SET_COMPARE_INVERT_0, PWM_NORMAL);
```

The PWM compare output 0 is driven to high state when the value of the PWM Value Register 0 is greater than the value of the PWM counter.

5.14.3.63 PWM_SET_ACTIVE_HIGH_FAULTS - set selected FAULT inputs as active-high

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_ACTIVE_HIGH_FAULTS,
           Word16 param);
```

Arguments:

Table 5-520. PWM_SET_ACTIVE_HIGH_FAULTS ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	Parameter to select the fault pins. Use combination of the constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_SET_ACTIVE_HIGH_FAULTS* ioctl command controls the polarity of the FAULT input signals. The FAULT signals referenced by the parameter are configured as active-high.

This command modifies corresponding FPOLx bits in the PWM Fault Control Register (PMFCTL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_SET_ACTIVE_HIGH_FAULTS* ioctl command is implemented as a macro.

Example 5-464. PWM_SET_ACTIVE_HIGH_FAULTS

```
ioctl(PWM, PWM_SET_ACTIVE_HIGH_FAULTS, PWM_FAULT_0 |
      PWM_FAULT_1);
```

The PWM FAULT inputs 0 and 1 are configured as active-high.

5.14.3.64 PWM_SET_ACTIVE_LOW_FAULTS - set selected FAULT inputs as active-low

Call(s):

```
void ioctl(const int *pModuleBase, PWM_SET_ACTIVE_LOW_FAULTS,
           Word16 param);
```

Arguments:

Table 5-521. PWM_SET_ACTIVE_LOW_FAULTS ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	Parameter to select the fault pins. Use combination of the constants: PWM_FAULT_0 / PWM_FAULT_1 / PWM_FAULT_2 / PWM_FAULT_3

Description: The *PWM_SET_ACTIVE_LOW_FAULTS* ioctl command controls the polarity of the FAULT input signals. The FAULT signals referenced by the parameter are configured as active-low.

This command modifies corresponding FPOLx bits in the PWM Fault Control Register (PMFCTL).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_SET_ACTIVE_LOW_FAULTS* ioctl command is implemented as a macro.

Example 5-465. PWM_SET_ACTIVE_LOW_FAULTS

```
ioctl(PWM, PWM_SET_ACTIVE_LOW_FAULTS, PWM_FAULT_2);
```

The PWM FAULT input 2 is configured as active-low.

5.14.3.65 PWM_DISABLE_SYNC - disable SYNC feature on FAULT2 pin

Call(s):

```
void ioctl(const int *pModuleBase, PWM_DISABLE_SYNC, NULL);
```

Arguments:

Table 5-522. PWM_DISABLE_SYNC ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.

Description: The *PWM_DISABLE_SYNC* ioctl command restores the default behavior of the FAULT2 pin by disabling SYNC input or SYNC output functionality which was previously set by one of the PWM_ENABLE_SYNC_OUT or PWM_ENABLE_SYNC_IN ioctl commands.

This command writes zero to PWM Synchronization Window Register (SYNC), i.e. it clears the SYNC_OUT_EN bit and also the SYNC_WINDOW bit-field.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_DISABLE_SYNC* ioctl command is implemented as a macro.

Example 5-466. PWM_DISABLE_SYNC

```
ioctl(PWM, PWM_DISABLE_SYNC, NULL);
```

This code disables the SYNC feature and reverts the FAULT2 pin to the default functionality.

5.14.3.66 PWM_ENABLE_SYNC_OUT - enable SYNC_OUT signal on FAULT2 pin

Call(s):

```
void ioctl(const int *pModuleBase, PWM_ENABLE_SYNC_OUT, NULL);
```

Arguments:

Table 5-523. PWM_ENABLE_SYNC_OUT ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.

Description: The *PWM_ENABLE_SYNC_OUT* ioctl command replaces the PWM Fault 2 input signal with the PWM SYNC_OUT output pulse signal. This signal can be used to synchronize external circuitry with an internal PWM counter value. The pin still needs to be configured as "PWM Fault 2" pin in the SYS and GPIO modules.

This command writes value of 0x8000 to the PWM Synchronization Window Register (SYNC), i.e. it sets the SYNC_OUT_EN bit while clearing the SYNC_WINDOW bit-field.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_ENABLE_SYNC_OUT* ioctl command is implemented as a macro.

Example 5-467. PWM_ENABLE_SYNC_OUT

```
ioctl(PWM, PWM_ENABLE_SYNC_OUT, NULL);
```

This code enables the SYNC-output feature which replaces the PWM FAULT2 signal.

5.14.3.67 PWM_ENABLE_SYNC_IN - enable SYNC_IN input on FAULT2 pin

Call(s):

```
void ioctl(const int *pModuleBase, PWM_ENABLE_SYNC_IN,
     UWord16 param);
```

Arguments:

Table 5-524. PWM_ENABLE_SYNC_IN ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	UWord16 in the range 1...0x7fff representing the threshold value for the PWM counter and defining the synchronization window.

Description: The *PWM_ENABLE_SYNC_IN* ioctl command replaces the PWM Fault 2 input signal with the PWM SYNC_IN input signal. This signal can be used by an external circuitry to reset the internal PWM counter. The synchronization "window", in which the synchronization signal is only accepted, is specified as parameter.

For center aligned operation if the value of the PWM counter is less than the parameter value, the external synchronization is enabled. For a parameter value of 0x7FFF, the external sync can occur at any time.

For edge aligned operation if the value of the PWM counter is less than the parameter value or if the difference between the value of the PWM counter and the counter modulo value is less than the parameter value, then the external synchronization is enabled.

To achieve synchronization functionality, the pin still needs to be configured as "PWM Fault 2" pin in the SYS and GPIO modules.

This command writes the 15-bit parameter value to the PWM Synchronization Window Register (SYNC), i.e. it clears the SYNC_OUT_EN bit and sets the SYNC_WINDOW bit-field as a parameter value.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_ENABLE_SYNC_IN* ioctl command is implemented as a macro.

Example 5-468. PWM_ENABLE_SYNC_IN

```
ioctl(PWM, PWM_ENABLE_SYNC_IN, 0x7fff);
```

This code enables the SYNC-input synchronization signal on the Fault 2 pin to clear the internal PWM counter at any time.

5.14.3.68 PWM_WRITE_FILTx_REG - write the PWM Fault Filter Register

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WRITE_FILT0_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT1_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT2_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT3_REG,
     UWord16 param);
```

Arguments:

Table 5-525. PWM_WRITE_FILTx_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	UWord16 register value.

Description: The *PWM_WRITE_FILTx_REG* ioctl commands write directly to an appropriate PWM Fault Filter Register. On MC56F802x/3x, each of four Fault inputs can be filtered by applying a digital "counting" filter. The FFILTx registers contain all necessary bits and bit-fields to configure the filter, so these commands may be used to reconfigure this feature in run-time. However, the filters are typically configured statically (e.g. by using the Graphical Configuration Tool).

Each of these four commands writes the parameter value directly to an appropriate PWM Fault Filter Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_WRITE_FILTx_REG* ioctl command is implemented as a macro.

Example 5-469. PWM_WRITE_FILTx_REG

```
ioctl(PWM, PWM_WRITE_FILT0_REG, 0x0501);
```

This code configures the Fault 0 input filter in terms that the glitches shorter than 8 clocks are filtered-out. On the other hand, this filter causes 10 clocks latency for the Fault 0 input signal.

5.14.3.69 PWM_WRITE_FILTx_REG - write the PWM Fault Filter Register

Call(s):

```
void ioctl(const int *pModuleBase, PWM_WRITE_FILT0_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT1_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT2_REG,
     UWord16 param);
void ioctl(const int *pModuleBase, PWM_WRITE_FILT3_REG,
     UWord16 param);
```

Arguments:

Table 5-526. PWM_WRITE_FILTx_REG ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	UWord16 register value.

Description: The *PWM_WRITE_FILTx_REG* ioctl commands write directly to an appropriate PWM Fault Filter Register. On MC56F802x/3x, each of four Fault inputs can be filtered by applying a digital "counting" filter. The FFILTx registers contain all necessary bits and bit-fields to configure the filter, so these commands may be used to reconfigure this feature in run-time. However, the filters are typically configured statically (e.g. by using the Graphical Configuration Tool).

Each of these four commands writes the parameter value directly to an appropriate PWM Fault Filter Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PWM_WRITE_FILTx_REG* ioctl command is implemented as a macro.

Example 5-470. PWM_WRITE_FILTx_REG

```
ioctl(PWM, PWM_WRITE_FILT0_REG, 0x0501);
```

This code configures the Fault 0 input filter in terms that the glitches shorter than 8 clocks are filtered-out. On the other hand, this filter causes 10 clocks latency for the Fault 0 input signal.

5.14.3.70 PWM_SET_PULSE_EDGE_CONTROL_x - PWM pair pulse edge control

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
       PWM_SET_PULSE_EDGE_CONTROL_0, UWord16 param);
UWord16 ioctl(const int *pModuleBase,
       PWM_SET_PULSE_EDGE_CONTROL_1, UWord16 param);
UWord16 ioctl(const int *pModuleBase,
       PWM_SET_PULSE_EDGE_CONTROL_2, UWord16 param);
```

Arguments:

Table 5-527. PWM_SET_PULSE_EDGE_CONTROL_x ioctl call arguments

*pModuleBase	in	The PWM module identifier. Use PWM on MC56F80xx.
param	in	Parameter to select pulse edge control. Use one of the predefined constants: PWM_NORMAL_MODE / PWM_PWMVAL_MODE

Description: The *PWM_SET_PULSE_EDGE_CONTROL_x* ioctl commands sets pulse edge control. The `PWM_SET_PULSE_EDGE_CONTROL_0` `ioctl` `command` `controls` `PWM0/PWM1` `pair.` The `PWM_SET_PULSE_EDGE_CONTROL_1` `ioctl` `command` `controls` `PWM2/PWM3` `pair.` The `PWM_SET_PULSE_EDGE_CONTROL_2` `ioctl` `command` `controls` `PWM4/PWM5` `pair.` This ioctl command sets normal operation, when PWM_NORMAL_MODE is used as a parameter or PWM value activated mode, when PWM_PWMVAL_MODE is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *PWM_SET_PULSE_EDGE_CONTROL_x* ioctl command is implemented as a macro.

Example 5-471. PWM_SET_PULSE_EDGE_CONTROL_x

```
ioctl(PWM, PWM_SET_PULSE_EDGE_CONTROL_0, PWM_NORMAL_MODE);
```

This code sets PWM0/PWM1 pair to normal mode.

5.14.4 PWM Driver Application

The PWM driver application is designed for Freescale/Motorola EVMs. It is intended to illustrate the usage of this driver by a real example and also to verify the functionality by blinking LEDs associated to the PWM pins. The general purpose LED, associated with the GPIO pin, indicates a PWM Reload Interrupt and changes its state within the interrupt service routine. The PWM driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\pwm_demo* and consists of the application *pwm_demo.mcp* and the source code for the application *main.c*. To view the signal apply a scope probe.

Example 5-472. PWM Driver Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Thu, 08/Feb/2007, 16:37:11
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 4 MHz
    VCO frequency: 240 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0081
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
```

```
                          TMR A: Enable , TMR B: Enable , TMR C: Enable
                          TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                          CAN: Enable , ADC A: Enable , ADC B: Enable
                          EMI: Enable
.*/
#define SIM_GPS_INIT                    0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                0x0005
#define SEMI_CSBAR1_INIT                0x0005
#define SEMI_CSBAR2_INIT                0x0005
#define SEMI_CSOR0_INIT                 0x1FC3
#define SEMI_CSOR1_INIT                 0x1BA3
#define SEMI_CSOR2_INIT                 0x1DA3

/*.
    INTC Configuration
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0000
#define INT_VECTOR_ADDR_78              pwm_reload_isr
#define INT_PRIORITY_LEVEL_78           INTC_LEVEL1

/*.
    PWM_A Configuration
-----------------------------------------
    PWM module operation: Disabled
    Prescaler: /8 , PWM clock period: 2 us
    PWM frequency: 7.62963 Hz , Period: 131.06801 ms
    PWM Deadtime: 8.18975 ms
    PWM Reload Frequency: Every 10 opportunity
    Alignment: Center
    Debug Mode Operation: Stop
```

```
    Wait Mode Operation: Stop
    Deadtime Correction Method: Manual correction (no correction)
    Half Cycle Reload: Disabled
    Keep Hardware Acceleration Register Bits Writable: Disabled
    Output Pad Enable: Enabled
    Load OK: Yes
    Swap & Mask Mode: DSP56F80X compatible
    Write Protection: No
    Top Polarity   : Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Bottom Polarity: Channels 0-1: Positive
                     Channels 2-3: Positive
                     Channels 4-5: Positive
    Chann. coupling: Channels 4-5: Complementary
                     Channels 2-3: Complementary
                     Channels 0-1: Complementary
    Clearing Mode:   Fault 0 pin: Manual
                     Fault 1 pin: Manual
                     Fault 2 pin: Manual
    Asymmetric Operation: Channels 0-1: Off (Correction Method used only)
                          Channels 2-3: Off (Correction Method used only)
                          Channels 4-5: Off (Correction Method used only)
.*/
#define PWM_A_PMCTL_INIT                  0x90E2
#define PWM_A_PMOUT_INIT                  0x8000
#define PWM_A_PWMCM_INIT                  0x7FFF
#define PWM_A_PMDEADTM_INIT               0x0FFF
#define PWM_A_PMDISMAP1_INIT              0x0000
#define PWM_A_PMDISMAP2_INIT              0x0000

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  9: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin 10: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
.*/
#define GPIO_C_DDR_INIT                   0x000F
#define GPIO_C_PER_INIT                   0x00F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  7: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
```

```
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_DDR_INIT                 0x00C0
#define GPIO_D_PER_INIT                 0x1F01


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-473. PWM Driver Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application demonstrating the use of PWM driver
*
* TARGET: MC56F83xx devices
*
*****************************************************************************/

#include "qs.h"

#include "sys.h"
#include "cop.h"
#include "intc.h"
#include "gpio.h"
#include "pwm.h"

/* board-specific LEDs */
#include "../board.h"

/* forward prototypes */
void device_init(void);


/*
 *  The main
 */

int main(void)
{
    pwm_sComplementaryValues comp;

    /* initialize SYS, COP and pins */
    device_init();

    /* make sure the RED LED is GPIO output */
    ioctl(GPIO_LEDS, GPIO_SETAS_GPIO, LED_R);
    ioctl(GPIO_LEDS, GPIO_SETAS_OUTPUT, LED_R);
    ioctl(GPIO_LEDS, GPIO_CLEAR_PIN, LED_R);

    /* configure PWM_A module */
    ioctl(PWM_A, PWM_INIT, NULL);

    /* sets 50% duty cycle for all channels */
    comp.pwmChannel_0_Value = 0x4000;
    comp.pwmChannel_2_Value = 0x4000;
    comp.pwmChannel_4_Value = 0x4000;

    /* calculates the actual contents of the value registers, fills these registers,
sets Load OK bit */
    ioctl(PWM_A, PWM_UPDATE_VALUE_REGS_COMPL, &comp);

    /* enable PWM_A */
    ioctl(PWM_A, PWM_DEVICE, PWM_ENABLE);
```

```
    /* configure Interrupt Controller (IPR) and enable interrupts */
    ioctl(INTC, INTC_INIT, NULL);
    archEnableInt();

    while(1)
    {
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
    }
}

/*
 *  An example of reload interrupt service routine
 */

#pragma interrupt on

void pwm_reload_isr(void)
{
  /* clears reload interrupt flag */
  ioctl(PWM_A, PWM_CLEAR_RELOAD_FLAG, NULL);

  /* toggle the RED LED */
  ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_R);
}

#pragma interrupt off


/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.15 SCI Driver

This section describes the API for the MC56F83xx and the MC56F80xx Serial Communication Interface (SCI). The functionality of the SCI module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual**, **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.15.1 Introduction

There are either one or two SCI modules on each 56F8xxx family member. SCI pins are always shared with GPIO pins, so the GPIO module must be configured to enable peripheral functionality of all SCI pins used.

Each SCI module has a 13-bit baud rate selection (16-bit on MC56F802x/3x and MC56F800x), programmable 8-bit or 9-bit data format and hardware parity checking. Four interrupt requests can be generated by the SCI module (Receiver Full, Receiver Error, Transmitter Empty, Transmitter Idle).

The MC56F802x/3x family newly adds a support for a queued operation of SCI (receive and transmit FIFOs), enabling to lower the interrupt rate to CPU and to simplify software design in some scenarios. The SCI module with such a queued capability is often referred as QSCI. However, the module identifier used in the low-level drivers of DSP56800E_Quick_Start and in *appconfig.h* initialization values remains SCI, SCI_0 or SCI_1 for backward code compatibility.

5.15.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.15.3.

Table 5-528. SCI Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F8037	MC56F83xx
SCI 0 (SCI0_BASE)	0xF0E0	0xF0B0	0xF200	0xF200	0xF280
SCI 1 (SCI1_BASE)	N/A	N/A	N/A	0xF210	0xF290

5.15.2.1 API Definition

The following header files are needed in order to use the SCI device driver:

Required Header File(s):

```
#include "qs.h"
#include "sci.h"
```

Public Data Structure(s):

N/A

5.15.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static SCI module configuration by the driver initialization routine. Symbol *x* in the table should be replaced with the number of the SCI module e.g. SCI_0_SCIBR_INIT. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-523 for more details.

Table 5-529. SCI Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
SCI*x*CHAR	type name char or UWord16	Specifies the type of single 'character' for buffer operations. The 'UWord16' is default and is backward compatible with HawkV1 implementation. The 'char' is the optimal choice for 56F800E family as it supports arrays or byte-wide elements.
SCI_*x*_SCIBR_INIT	UWord16	Initial value of the SCI Baud Rate Register.
SCI_*x*_SCICR_INIT	UWord16	Initial value of the SCI Control (Control 1) Register.
SCI_*x*_SCICR2_INIT	UWord16	Initial value of the SCI Control 2 Register.
SCI_CORE_FREQ	UWord16	This value specifies the SCI clock frequency in Hz. When this macro is not defined, the frequency is calculated from OCCS settings. The *occs.h* file should be included before the *sci.h*.

In addition to standard configuration values described above, the DSP56800E_Quick_Start tool contains also SCI software interface which enables buffered operations in various configurations. To enable this built-in software layer, the following configuration items have to be specified in the *appconfig.h* file.

Table 5-530. SCI Operation Items for appconfig.h

SYMBOL	TYPE / VALUE	DESCRIPTION
INT_VECTOR_ADDR_yy yy = SCI Receiver Full Interrupt	Sci0RxFullISR, Sci1RxFullISR	Name of the built in interrupt service routine implemented in SCI driver.
INT_VECTOR_ADDR_yy yy = SCI Receiver Error Interrupt	Sci0RxErrorISR, Sci1RxErrorISR	Name of the built in interrupt service routine implemented in SCI driver.
INT_VECTOR_ADDR_yy yy = SCI Transmitter Empty Interrupt	Sci0TxEmptyISR, Sci1TxEmptyISR	Name of the built in interrupt service routine implemented in SCI driver.
INT_PRIORITY_LEVEL_yy yy = the three SCI interrupts as described above	INTC_LEVEL0 | INTC_LEVEL1 | INTC_LEVEL2	To enable the SCI interrupts, they must be assigned a interrupt priority level.

The built-in software layer can operate in two different modes, depending on the SCI_x_HIGHLEVEL_MODE macro value. When high level layer is to be used on more than one SCI module, the same mode must be selected for all modules.

- **BLOCKING / NON_BLOCKING mode** - this implementation is backward compatible with previous (HawkV1) implementation and enables basic transmission and reception of prepared buffers. This mode is implemented in the *scibl.c* file.
- **BUFFERED mode** - which uses internal circular buffer for continuous receive/transmit operations. The *read* and *write* calls just access this internal buffer. This mode also contains optional XON/XOFF or RTS/CTS protocol implementation on the slave side. This mode is implemented in the *scibf.c* file.

There are several *appconfig.h* configuration items for the built in software layer:

Table 5-531. Built-in Software Layer Configuration Items for appconfig.h

SYMBOL	TYPE / VALUE	DESCRIPTION
SCI_x_HIGHLEVEL_MODE	0 | 1 | 2	Enables or disables the high level software layer for SCI module or modules. 0 .. no software layer is compiled 1 .. BLOCKING/NON_BLOCKING mode 2 .. BUFFERED mode
SCI_*x*_RX_CHAR_HOOK	function name declared as: `UWord16 hook(UWord16 rxchar)`	The name of callback function which is called after each character received. Return nonzero to discard the received character.
SCI_*x*_RX_ERROR_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when any error is detected by the receiver.
SCI_*x*_TX_CHAR_HOOK	function name declared as: `UWord16 hook(UWord16 txchar)`	The name of callback function which is called after each character transmitted. Return nonzero when custom (asynchronous) character was inserted to the SCI transmit buffer.
SCI_*x*_RX_FINISHED_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when NON_BLOCKING read operation finishes,
SCI_*x*_TX_FINISHED_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when NON_BLOCKING write operation finishes,
SCI_*x*_RX_LOWBUFFER_HOOK	function name declared as: `void hook(bool bIsFull)`	The name of callback function which is called when circular buffer is near to be full during BUFFERED operation. The “bIsFull” is “true” when buffer just get full or “false” when it is getting free again.

When BUFFERED operation is used (SCI_x_HIGHLEVEL_MODE defined as 2), the following configuration items also apply:

Table 5-532. Buffer Space Monitoring Configuration Items

SYMBOL	TYPE / VALUE	DESCRIPTION
SCI_*x*_RX_BUFFER_PTR	pointer to array of SCI0CHAR or SCI1CHAR elements	The buffer which will be used as internal circular buffer for BUFFERED operation. When not defined, the buffer is allocated as static array directly by the driver.
SCI_*x*_RX_BUFFER_SIZE	UWord16	The size of the buffer to allocate or specified by SCI_x_RX_BUFFER_PTR.
SCI_*x*_RX_BUFFER_LOWLIMIT	UWord16	When the free space in the buffer sinks bellow this number, the buffer-low events are triggered.
SCI_*x*_RX_BUFFER_OKLIMIT	UWord16	When the free space in the buffer grows above this number, the buffer-ok events are triggered. When not defined, the value equal to SCI_x_RX_BUFFER_LOWLIMIT is taken. When defined, it must be equal or greater than SCI_x_RX_BUFFER_LOWLIMIT.
SCI_*x*_RX_XONXOFF	n/a	When defined, the software XON/XOFF protocol is enabled on this (slave) side. The buffer limit values must be defined to enable XON/XOFF protocol.
SCI_*x*_XONCHAR	UWord8	The value of XON character. When not defined, the standard 0x11 is used.
SCI_*x*_XOFFCHAR	UWord8	The value of XOFF character. When not defined, the standard 0x13 is used.

5.15.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
void ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);

void read (const int *pModuleBase, const int mode,
           int *address, int size);
void write(const int *pModuleBase, const int mode,
           int *address, int size);
```

Description: The *ioctl* call "changes" the SCI device modes or accesses the SCI register(s). The *read* or *write* calls call functions for block read from or block write to the SCI module.

Arguments:

Table 5-533. SCI Driver Arguments - ioctl

pModuleBase	in	SCI module identifier. Use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
cmd	in	Commands found in *sci.h* which are used to modify the SCI module status and control registers. See Table 5-535.
param	in, inout	Used to pass the relevant data to ioctl function call.

Table 5-534. SCI Driver Arguments - read/write

pModuleBase	in	SCI module identifier. Use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
mode	in	Used to set the read/write functions mode. Use NON_BLOCKING or BLOCKING.
address	in, inout	Address of allocated data buffer.
size	in	Number of words to be read (in case of read function call) or to be written (in case of write function call).

Item Separator Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-535. ioctl commands

Cmd	param	Return	Description
SCI_INIT	NULL	None	Initializes selected SCI module by data from configuration file (app-config.h).
SCI_SET_BAUDRATE	UWord16 SCI_BAUD_xx values	None	Configures the SCI Baud Rate Register. The SCI_BAUD_xx (xx = baud rate) values are defined for most common clock frequencies.
SCI_OPERATING_MODE	SCI_INTERNAL_LOOP_MODE / SCI_NORMAL_MODE / SCI_SINGLE_WIRE_MODE	None	Sets SCI loop mode operation.
SCI_TRANSMITTER	SCI_ENABLE / SCI_DISABLE	None	Enables or disables SCI Transmitter.
SCI_RECEIVER	SCI_ENABLE / SCI_DISABLE	None	Enables or disables SCI Receiver.
SCI_WAKEUP_CONDITION	SCI_WAKE_BY_ADDRESS / SCI_WAKE_BY_IDLE	None	Sets SCI wake up mode.
SCI_DATA_FORMAT	SCI_WORD_9BIT / SCI_WORD_8BIT	None	Sets SCI word length.
SCI_PARITY	SCI_PARITY_ODD / SCI_PARITY_EVEN / SCI_PARITY_NONE	None	Sets SCI parity function mode.
SCI_DATA_POLARITY	SCI_INVERTED / SCI_NOT_INVERTED	None	Sets polarity of transmitted and received data.
SCI_STOP_IN_WAIT	SCI_ENABLE / SCI_DISABLE	None	Enables or disables the low-power STOP mode during CPU WAIT mode.
SCI_SEND_BREAK	NULL	None	Sends a single break character.
SCI_WAIT	NULL	None	Put SCI receiver in sleep mode.
SCI_WAKEUP	NULL	None	Wake up SCI receiver.
SCI_INT_ENABLE	SCI_TX_EMPTY | SCI_TX_IDLE | SCI_RX_FULL | SCI_RX_ERROR |	None	Enables selected SCI interrupts.
SCI_INT_DISABLE		None	Disables selected SCI interrupts.
SCI_TX_EMPTY_INT *SCI_TX_IDLE_INT* *SCI_RX_FULL_INT* *SCI_RX_ERROR_INT*	*SCI_ENABLE / SCI_DISABLE*	*None*	*Interrupt enable or disable for individual interrupt sources. The same operation possible with SCI_INT_ENABLE / DISABLE*
only on MC56F80xx**:** SCI_LIN_MODE	SCI_ENABLE / SCI_DISABLE	None	Enables or disables LIN slave operation.

Table 5-535. ioctl commands (Continued)

Cmd	param	Return	Description
SCI_READ_DATA	NULL	UWord16	Reads SCI Data Register.
SCI_WRITE_DATA	UWord16	None	Writes to SCI Data Register.
SCI_READ_CONTROL_REG	NULL	UWord16	Reads SCI Control Register.
SCI_WRITE_CONTROL_REG	UWord16	None	Writes to SCI Control Register.
SCI_GET_STATUS_REG	NULL	UWord16	Reads SCI Status Register.
SCI_CLEAR_STATUS_REG	NULL	None	Clears SCI Status Register.
SCI_TEST_STATUS_REG	UWord16, combination of status register bits: SCI_TX_EMPTY_FLAG | SCI_TX_IDLE_FLAG | SCI_RX_FULL_FLAG | SCI_RX_IDLE_LINE_FLAG | SCI_OVERRUN_FLAG | SCI_NOISE_FLAG | SCI_FRAMING_ERROR_FLAG | SCI_PARITY_ERROR_FLAG | SCI_RX_ACTIVE_FLAG | **only on** MC56F80xx**:** SCI_LIN_SYNC_ERROR_FLAG	UWord16	Tests bits in SCI Status Register.
SCI_GET_TX_EMPTY *SCI_GET_TX_IDLE* *SCI_GET_RX_FULL* *SCI_GET_RX_IDLE* *SCI_GET_RX_OVERRUN* *SCI_GET_RX_NOISE_ERROR* *SCI_GET_RX_FRAMING_ERROR* *SCI_GET_RX_PARITY_ERROR* *SCI_GET_RX_ACTIVE* **only on** MC56F80xx**:** *SCI_GET_LIN_SYNC_ERROR*	*NULL*	*UWord16*	*Single-bit testing in the SCI Status Register. Returns non-zero if the particular bit is set.* *These commands can be used instead of the command above when testing a single status bit.*
SCI_GET_ERROR	NULL	UWord16	Gets SCI Status Register OR, NF, FE and PF bits (Overrun Flag, Noise Flag, Framing Error Flag, Parity Error Flag). Returns non-zero when any of the bits is set.
only on MC56F802x/3x: SCI_QUEUED_MODE	SCI_ENABLE / SCI_DISABLE	None	Enables the SCI receive and transmit FIFOs.
only on MC56F802x/3x: SCI_SET_TXEMPTY_CONDITION	SCI_TXEMPTY_*condition* where *condition* is one of: WHEN_FIFO_EMPTY / WHEN_1_OR_LESS_IN_FIFO / WHEN_2_OR_LESS_IN_FIFO / WHEN_3_OR_LESS_IN_FIFO	None	Sets transmit FIFO threshold when a new characters will be requested from the CPU (by signaling the TXEMPTY interrupt).

Table 5-535. ioctl commands (Continued)

Cmd	param	Return	Description
only on MC56F802x/3x: SCI_SET_RXFULL_CONDITION	SCI_RXFULL_*condition* where *condition* is one of: WHEN_1_OR_MORE_IN_FIFO WHEN_2_OR_MORE_IN_FIFO WHEN_3_OR_MORE_IN_FIFO WHEN_4_OR_MORE_IN_FIFO	None	Sets the receive FIFO threshold when a new characters will be requested from the CPU (by signaling the TXEMPTY interrupt).
only on MC56F802x/3x: SCI_CAN_READ_DATA	NULL	UWord16	Returns non-zero if valid data are waiting in the receive FIFO.
only on MC56F802x/3x: SCI_CAN_WRITE_DATA	NULL	UWord16	Returns non-zero if there is a free space in the transmit FIFO.
High-level software layer commands:			
SCI_CLEAR_EXCEPTION	NULL	None	Clears read/write functions exception if exists.
SCI0_GET_STATUS	NULL	UWord16	Get read/write functions SCI0 operation status.
SCI1_GET_STATUS	NULL	UWord16	Get read/write functions SCI1 operation status.
SCI_WRITE_CANCEL	NULL	None	Cancels non-blocking write operation.
SCI_READ_CANCEL	NULL	None	Cancels non-blocking read operation.
SCI_BUFFERED_RX	SCI_ENABLE / SCI_DISABLE	None	Enables the BUFFERED read/write operations. The buffer pointers are initialized and interrupts enabled.
SCI_BUFFERED_TX			
SCI_GET_RX_CHARS_READY	NULL	Word16	Number of characters pending in the receive buffer during the BUFFERED operation.
SCI_GET_RX_BUFFER_ FREESPACE	NULL	Word16	Returns the free space in the receive buffer during the BUFFERED operation.
SCI_GET_TX_BUFFER_ FREESPACE	NULL	Word16	Returns the free space in the transmit buffer during the BUFFERED operation.
SCI_SEND_XON	NULL	None	Send the XON priority character.
SCI_SEND_XOFF	NULL	None	Send the XOFF priority character.

Table 5-536. read/write modes

Mode	Return	Description
NON_BLOCKING	None	Calls read/write function in non-blocking mode. Internal functions of the SCI driver are used to read/write data. See Chapter 5.15.3.39 and Chapter 5.15.3.40.
BLOCKING	None	Calls read/write function in blocking mode. Internal functions of the SCI driver are used. See Chapter 5.15.3.39 and Chapter 5.15.3.40.
BUFFERED	None	Calls read/write function in buffered mode. Internal functions of the SCI driver are used. See Chapter 5.15.3.39 and Chapter 5.15.3.40.

5.15.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.15.3.1 SCI_INIT - initialize SCI module

Call(s):

```
void ioctl(const int *pModuleBase, SCI_INIT, NULL);
```

Arguments:

Table 5-537. SCI_INIT ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_INIT* ioctl command initializes the SCI module by writing to the proper registers. These symbols are intended for the application (project) specific configuration file *appconfig.h*.

Returns: None.

Range Issues: None.

Special Issues: The *SCI_INIT* ioctl command must be used if the *read*/*write* functions are called within the application.

Design/Implementation: The *SCI_INIT* ioctl command is implemented as a function call.

Example 5-474. SCI_INIT

```
ioctl(SCI_0, SCI_INIT, NULL);
```

This code initializes the SCI 0.

5.15.3.2 SCI_SET_BAUDRATE - writes to SCI Baud Rate Register

Call(s):

```
void ioctl(const int *pModuleBase, SCI_SET_BAUDRATE,
           UWord16 param);
```

Arguments:

Table 5-538. SCI_SET_BAUDRATE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Module clock divisor which is written to 13-bit SCI Baud Rate Register.

Description: The *SCI_SET_BAUDRATE* ioctl command writes to the 13-bit SCI Baud Rate Register. The SCI module clock is generated from the core clock using the SCI baud rate divisor from the SCI Baud Rate Register.

The special constants SCI_BAUD_xx are defined in *sci.h* for the most common SCI clock frequencies. For example when SCI_CORE_FREQ is defined or calculated as 52 MHz, the SCI_BAUD_9600 is automatically defined as 339 giving the baud rate 9587 bps, which is the nearest possible value to the 9600. When the clock setting changes to e.g. 4 MHz, the SCI_BAUD_9600 is redefined to 26, which gives also the nearest value to the 9600 bps (9615).

Returns: None.

Range Issues: *param* must be set within 0 and 8191, due to the 13-bit SCI Baud Rate Register.

Special Issues: None.

Design/Implementation: The *SCI_SET_BAUDRATE* ioctl command is implemented as a macro.

Example 5-475. SCI_SET_BAUDRATE

```
ioctl(SCI_0, SCI_SET_BAUDRATE, SCI_BAUD_9600);
```

This code sets a baud rate of 9600 bps.

5.15.3.3 SCI_OPERATING_MODE - sets SCI loop mode operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_OPERATING_MODE,
     UWord16 param);
```

Arguments:

Table 5-539. SCI_OPERATING_MODE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use the following constants: SCI_INTERNAL_LOOP_MODE SCI_NORMAL_MODE SCI_SINGLE_WIRE_MODE

Description: The *SCI_OPERATING_MODE* ioctl command sets the SCI module to normal operation (SCI_NORMAL_MODE), loop mode with internal TXD fed back to RXD (SCI_INTERNAL_LOOP_MODE) or single-wire loop mode with TXD output fed back to RXD (SCI_SINGLE_WIRE_MODE).

The *SCI_OPERATING_MODE* command modifies the LOOP (Loop Select Bit) and the RSRC (Receiver Source Bit) bits in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_OPERATING_MODE* ioctl command is implemented as a macro.

Example 5-476. SCI_OPERATING_MODE

```
ioctl(SCI_0, SCI_OPERATING_MODE, SCI_SINGLE_WIRE_MODE);
```

This code sets the SCI 0 module to single-wire loop mode.

5.15.3.4 SCI_TRANSMITTER - enables or disables SCI transmitter

Call(s):

```
void ioctl(const int *pModuleBase, SCI_TRANSMITTER,
           UWord16 param);
```

Arguments:

Table 5-540. SCI_TRANSMITTER ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable SCI transmitter or SCI_DISABLE to disable SCI transmitter.

Description: The *SCI_TRANSMITTER* ioctl command enables or disables the SCI transmitter, modifying the TE (Transmitter Enable) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_TRANSMITTER* ioctl command is implemented as a macro.

Example 5-477. SCI_TRANSMITTER

```
ioctl(SCI_0, SCI_TRANSMITTER, SCI_ENABLE);
```

This code enables the SCI 0 transmitter.

5.15.3.5 SCI_RECEIVER - enables or disables SCI receiver

Call(s):

```
void ioctl(const int *pModuleBase, SCI_RECEIVER, UWord16 param);
```

Arguments:

Table 5-541. SCI_RECEIVER ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable SCI receiver or SCI_DISABLE to disable SCI receiver.

Description: The *SCI_RECEIVER* ioctl command enables or disables the SCI receiver, modifying the RE (Receiver Enable) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_RECEIVER* ioctl command is implemented as a macro.

Example 5-478. SCI_RECEIVER

```
ioctl(SCI_0, SCI_RECEIVER, SCI_ENABLE);
```

This code enables the SCI 0 receiver.

5.15.3.6 SCI_WAKEUP_CONDITION - sets SCI wake up mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WAKEUP_CONDITION,
           UWord16 param);
```

Arguments:

Table 5-542. SCI_WAKEUP_CONDITION ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_WAKE_BY_ADDRESS to set SCI wake-up by address mode or SCI_WAKE_BY_IDLE to set SCI wake-up by idle mode

Description: The *SCI_WAKEUP_MODE* ioctl command sets which condition wakes up the SCI module. The SCI wake-up mode can be set to wake up according to address (a logic 1 in the most significant bit position of the received data character wakes up the SCI) or to wake up on idle (idle condition on the RXD pin wakes up the SCI). The *SCI_WAKEUP_MODE* ioctl command modifies the WAKE (Wake-up Condition Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_WAKEUP_MODE* ioctl command is implemented as a macro.

Example 5-479. SCI_WAKEUP_CONDITION

```
ioctl(SCI_0, SCI_WAKEUP_CONDITION, SCI_WAKE_BY_ADDRESS);
```

This code sets the SCI 0 wake-up by address mode.

5.15.3.7 SCI_DATA_FORMAT - sets SCI word length

Call(s):

```
void ioctl(const int *pModuleBase, SCI_DATA_FORMAT,
           UWord16 param);
```

Arguments:

Table 5-543. SCI_DATA_FORMAT ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_WORD_9BIT to set word length 9 bit or SCI_WORD_8BIT to set word length 8bit.

Description: The *SCI_DATA_FORMAT* ioctl command sets the word length for 8 bits (one start bit, eight data bits, one stop bit) or 9 bits (one start bit, nine data bits, one stop bit). The *SCI_DATA_FORMAT* ioctl command modifies the M (Data Format Mode Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_DATA_FORMAT* ioctl command is implemented as a macro.

Example 5-480. SCI_DATA_FORMAT

```
ioctl(SCI_0, SCI_DATA_FORMAT, SCI_WORD_9BIT);
```

This code sets the SCI 0 word length to 9 bit.

5.15.3.8 SCI_PARITY - sets SCI parity function mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_PARITY, UWord16 param);
```

Arguments:

Table 5-544. SCI_PARITY ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_PARITY_ODD for odd parity, SCI_PARITY_EVEN for even parity or SCI_PARITY_NONE to disable parity function.

Description: The *SCI_PARITY* ioctl command enables or disables the parity function of the SCI module. If enabled, the parity can be odd or even. The *SCI_PARITY* ioctl command modifies the PE (Parity Enable Bit) and the PT (Parity Type Bit) bits in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_PARITY* ioctl command is implemented as a macro.

Example 5-481. SCI_PARITY

```
ioctl(SCI_0, SCI_PARITY, SCI_PARITY_EVEN);
```

This code enables the parity function of the SCI 0 module and sets even parity.

5.15.3.9 SCI_DATA_POLARITY - sets polarity of transmitted and received data

Call(s):

```
void ioctl(const int *pModuleBase, SCI_DATA_POLARITY,
           UWord16 param);
```

Arguments:

Table 5-545. SCI_DATA_POLARITY ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_INVERTED to set inverted mode or SCI_NOT_INVERTED to set normal mode.

Description: The *SCI_DATA_POLARITY* ioctl command sets inverted (transmit and receive data bits are inverted) or normal (transmit and receive data bits are not inverted) mode. The *SCI_DATA_POLARITY* ioctl command modifies the POL (Polarity Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_DATA_POLARITY* ioctl command is implemented as a macro.

Example 5-482. SCI_DATA_POLARITY

```
ioctl(SCI_0, SCI_DATA_POLARITY, SCI_INVERTED);
```

This code sets inverted mode of the SCI 0.

5.15.3.10 SCI_STOP_IN_WAIT- sets SCI operation mode in Wait mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_STOP_IN_WAIT,
           UWord16 param);
```

Arguments:

Table 5-546. SCI_STOP_IN_WAIT ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use the following constants to enable or disable the low-power STOP mode during CPU Wait mode SCI_ENABLE to enable the low-power STOP mode in Wait SCI_DISABLE to disable the low-power STOP mode in Wait

Description: The *SCI_STOP_IN_WAIT* ioctl command enables or disables the SCI low-power STOP mode during Wait mode. The *SCI_STOP_IN_WAIT* ioctl command modifies the SWAI (Stop in Wait Mode) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_STOP_IN_WAIT* ioctl command is implemented as a macro.

Example 5-483. SCI_STOP_IN_WAIT

```
ioctl(SCI_0, SCI_STOP_IN_WAIT, SCI_ENABLE);
```

This code enables the SCI 0 low-power STOP mode of the SCI module in WAIT mode.

5.15.3.11 SCI_SEND_BREAK - sends one break character

Call(s):

```
void ioctl(const int *pModuleBase, SCI_SEND_BREAK, NULL);
```

Arguments:

Table 5-547. SCI_SEND_BREAK ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_SEND_BREAK* ioctl command sends one brake character by toggling the SBK (Send Break Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_* ioctl command is implemented as a macro.

Example 5-484. SCI_SEND_BREAK

```
ioctl(SCI_0, SCI_SEND_BREAK, NULL);
```

This code sends one break character from SCI 0.

5.15.3.12 SCI_WAIT - puts SCI receiver in sleep mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WAIT, NULL);
```

Arguments:

Table 5-548. SCI_WAIT ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_WAIT* ioctl command puts the SCI receiver in sleep mode by modifying the RWU (Receiver Wake-up Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_WAIT* ioctl command is implemented as a macro.

Example 5-485. SCI_WAIT

```
ioctl(SCI_0, SCI_WAIT, NULL);
```

This code puts the SCI 0 receiver in sleep mode.

5.15.3.13 SCI_WAKEUP - wakes up SCI receiver

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WAKEUP, NULL);
```

Arguments:

Table 5-549. SCI_WAKEUP ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_WAKEUP* ioctl command wakes up the SCI receiver by modifying the RWU (Receiver Wake-up Bit) bit in the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_WAKEUP* ioctl command is implemented as a macro.

Example 5-486. SCI_WAKEUP

```
ioctl(SCI_0, SCI_WAKEUP, NULL);
```

This code wakes up the SCI 0 receiver.

5.15.3.14 SCI_INT_ENABLE - enables SCI Interrupts

Call(s):

```
void ioctl(const int *pModuleBase, SCI_INT_ENABLE,
           UWord16 param);
```

Arguments:

Table 5-550. SCI_INT_ENABLE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use the combination of interrupt flags SCI_TX_EMPTY | SCI_TX_IDLE | SCI_RX_FULL | SCI_RX_ERROR

Description: The *SCI_INT_ENABLE* ioctl command enables selected SCI interrupts. The *SCI_INT_ENABLE* ioctl command modifies (sets) directly the appropriate bits in the SCI Control Register and SCI Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-487. SCI_INT_ENABLE

```
ioctl(SCI_0, SCI_INT_ENABLE, SCI_TX_EMPTY);
```

This code enables the SCI 0 Transmitter Empty Interrupt.

5.15.3.15 SCI_INT_DISABLE - disables SCI Interrupts

Call(s):

```
void ioctl(const int *pModuleBase, SCI_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-551. SCI_INT_DISABLE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use the combination of interrupt flags SCI_TX_EMPTY | SCI_TX_IDLE | SCI_RX_FULL | SCI_RX_ERROR

Description: The *SCI_INT_DISABLE* ioctl command disables selected SCI interrupts. The *SCI_INT_DISABLE* ioctl command modifies (clears) directly the appropriate bits in the SCI Control Register and SCI Control Register 2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-488. SCI_INT_DISABLE

```
ioctl(SCI_0, SCI_INT_DISABLE, SCI_TX_EMPTY);
```

This code disables the SCI 0 Transmitter Empty Interrupt.

5.15.3.16 SCI_GET_STATUS_REG - reads SCI Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_STATUS_REG, NULL);
```

Arguments:

Table 5-552. SCI_GET_STATUS_REG ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_STATUS_REG* ioctl command reads the SCI Status Register.

Returns: Content of the SCI Status Register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_STATUS_REG* ioctl command is implemented as a macro.

Example 5-489. SCI_GET_STATUS_REG

```
tmp = ioctl(SCI_0, SCI_GET_STATUS_REG, NULL);
```

This code reads the content of the SCI 0 Status Register and stores it into variable *tmp*.

5.15.3.17 SCI_CLEAR_STATUS_REG - clears SCI Status Register

Call(s):

```
void ioctl(const int *pModuleBase, SCI_CLEAR_STATUS_REG, NULL);
```

Arguments:

Table 5-553. SCI_CLEAR_STATUS_REG ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_CLEAR_STATUS_REG* ioctl command performs a write operation to the SCI Status Register. To actually clear the SCI status register, the register must first be read using the *SCI_GET_STATUS_REGISTER* and then cleared by writing "any value" (not necessarily zero as used in this case) using the *SCI_CLEAR_STATUS_REG* command.

Note: To clear *SCI_TX_EMPTY_FLAG* or *SCI_RX_FULL_FLAG* flags, it is also necessary to write or read the SCI data register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_CLEAR_STATUS_REG* ioctl command is implemented as a macro.

Example 5-490. SCI_CLEAR_STATUS_REG

```
UWord16 stat = ioctl(SCI_0, SCI_GET_STATUS_REG, NULL);
...
...
ioctl(SCI_0, SCI_CLEAR_STATUS_REG, NULL);
```

This code clears the SCI 0 Status Register.

5.15.3.18 SCI_TEST_STATUS_REG - test bits in SCI Status Register

Call(s):

```
void ioctl(const int *pModuleBase, SCI_TEST_STATUS_REG,
     UWord16 param);
```

Arguments:

Table 5-554. SCI_TEST_STATUS_REG ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use the combination of status register bits: SCI_TX_EMPTY_FLAG | SCI_TX_IDLE_FLAG | SCI_RX_FULL_FLAG | SCI_RX_IDLE_LINE_FLAG | SCI_OVERRUN_FLAG | SCI_NOISE_FLAG | SCI_FRAMING_ERROR_FLAG | SCI_PARITY_ERROR_FLAG | SCI_RX_ACTIVE_FLAG | SCI_LIN_SYNC_ERROR_FLAG *(end-of-life parameters: SCI_SCISR_TDRE | SCI_SCISR_TIDLE | SCI_SCISR_RDRF | SCI_SCISR_RIDLE | SCI_SCISR_OR | SCI_SCISR_NF | SCI_SCISR_FE | SCI_SCISR_PF | SCI_SCISR_RAF).*

Description: The *SCI_TEST_STATUS_REG* ioctl command tests the selected bits of the SCI Status Register.

Returns: Non-zero value when at least one of the specified bits is set. Zero when all of the specified bits are clear.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_TEST_STATUS_REG* ioctl command is implemented as a macro.

Example 5-491. SCI_TEST_STATUS_REG

```
if(ioctl(SCI_0, SCI_TEST_STATUS_REG, SCI_RX_IDLE_LINE_FLAG))
{
      ...
}
```

This code tests the RX IDLE bit of the SCI 0 Status Register.

5.15.3.19 SCI_READ_DATA - reads SCI Data Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_READ_DATA, NULL);
```

Arguments:

Table 5-555. SCI_READ_DATA ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_READ_DATA* ioctl command reads the content of the SCI Data Register.

Returns: Content of the SCI Data Register.

Range Issues: Lower 8 bits of the 16 bit register are used, therefore the return value is comprised within 0x0000 and 0x00FF.

Special Issues: None.

Design/Implementation: The *SCI_READ_DATA* ioctl command is implemented as a macro.

Example 5-492. SCI_READ_DATA

```
tmp = ioctl(SCI_0, SCI_READ_DATA, NULL);
```

This code reads the content of the SCI 0 Data Register and stores it into variable *tmp*.

5.15.3.20 SCI_WRITE_DATA - writes to SCI Data Register

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WRITE_DATA, UWord16 param);
```

Arguments:

Table 5-556. SCI_WRITE_DATA ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	UWord16

Description: The *SCI_WRITE_DATA* ioctl command writes *param* to the SCI Data Register.

Returns: None.

Range Issues: Lower 8 bits of the 16 bit register are used, therefore the transmit data must be located in the lower byte of the word.

Special Issues: None.

Design/Implementation: The *SCI_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-493. SCI_WRITE_DATA

```
ioctl(SCI_0, SCI_WRITE_DATA, 'A');
```

This code writes 'A' to the SCI Data Register.

5.15.3.21 SCI_READ_CONTROL_REG - reads SCI Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_READ_CONTROL_REG,
              NULL);
```

Arguments:

Table 5-557. SCI_READ_CONTROL_REG ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_READ_CONTROL_REG* ioctl command reads the SCI Control Register.

Returns: Content of the SCI Control Register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-494. SCI_READ_CONTROL_REG

```
tmp = ioctl(SCI_0, SCI_READ_CONTROL_REG, NULL);
```

This code stores the content of the SCI Control Register into variable *tmp*.

5.15.3.22 SCI_WRITE_CONTROL_REG - writes to SCI Control Register

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WRITE_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-558. SCI_WRITE_CONTROL_REG ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	UWord16

Description: The *SCI_WRIRTE_CONTROL_REG* ioctl command writes *param* to the SCI Control Register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_WRIRTE_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-495. SCI_WRITE_CONTROL_REG

```
ioctl(SCI_0, SCI_WRITE_CONTROL_REG, 0x0004);
```

This code writes *0x0004* to the SCI 0 Control Register.

5.15.3.23 SCI_GET_TX_EMPTY - gets SCI Status Register TDRE bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_TX_EMPTY, NULL);
```

Arguments:

Table 5-559. SCI_GET_TX_EMPTY ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_TX_EMPTY* ioctl command reads the SCI Status Register TDRE (Transmit Data Register Empty Flag) bit.

Returns: *0* if the TDRE bit is cleared, non zero if the TDRE bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_TX_EMPTY* ioctl command is implemented as a macro.

Example 5-496. SCI_GET_TX_EMPTY

```
tmp = ioctl(SCI_0, SCI_GET_TX_EMPTY, NULL);
```

This code returns the SCI 0 Status Register, bit TDRE.

5.15.3.24 SCI_GET_TX_IDLE - gets SCI Status Register TIDLE bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_TX_IDLE, NULL);
```

Arguments:

Table 5-560. SCI_GET_TX_IDLE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_TX_IDLE* ioctl command reads the SCI Status Register TIDLE (Transmit Idle Flag) bit.

Returns: *0* if the TIDLE bit is cleared, non zero if the TIDLE bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_TX_IDLE* ioctl command is implemented as a macro.

Example 5-497. SCI_GET_TX_IDLE

```
tmp = ioctl(SCI_0, SCI_GET_TX_IDLE, NULL);
```

This code returns the SCI 0 Status Register, bit TIDLE.

5.15.3.25 SCI_GET_RX_FULL - gets SCI Status Register RDRF bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_FULL, NULL);
```

Arguments:

Table 5-561. SCI_GET_RX_FULL ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_FULL* ioctl command reads the SCI Status Register RDRF (Receiver Data Register Full Flag) bit.

Returns: *0* if the RDRF bit is cleared, non zero if the RDRF bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_FULL* ioctl command is implemented as a macro.

Example 5-498. SCI_GET_RX_FULL

```
tmp = ioctl(SCI_0, SCI_GET_RX_FULL, NULL);
```

This code returns the SCI 0 Status Register, bit RDRF.

5.15.3.26 SCI_GET_RX_IDLE - gets SCI Status Register RIDLE bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_IDLE, NULL);
```

Arguments:

Table 5-562. SCI_GET_RX_IDLE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_IDLE* ioctl command reads the SCI Status Register RIDLE (Receiver Idle Line Flag) bit.

Returns: *0* if the RIDLE bit is cleared, non zero if the RIDLE bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_IDLE* ioctl command is implemented as a macro.

Example 5-499. SCI_GET_RX_IDLE

```
tmp = ioctl(SCI_0, SCI_GET_RX_IDLE, NULL);
```

This code returns the SCI 0 Status Register, bit RIDLE.

5.15.3.27 SCI_GET_RX_ERROR - gets SCI Status Register OR, NF, FE and PF bits

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_ERROR, NULL);
```

Arguments:

Table 5-563. SCI_GET_RX_ERROR ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_ERROR* ioctl command reads the SCI Status Register OR, NF, FE and PF bits (Overrun Flag, Noise Flag, Framing Error Flag, Parity Error Flag).

Returns: *0* if the OR, NF, FE and PF bits are all cleared, non zero if any of the OR, NF, FE and PF bits is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_ERROR* ioctl command is implemented as a macro.

Example 5-500. SCI_GET_RX_ERROR

```
tmp = ioctl(SCI_0, SCI_GET_ERROR, NULL);
```

This code returns the SCI 0 Status Register, bits OR, NF, FE and PF.

5.15.3.28 SCI_GET_RX_OVERRUN - gets SCI Status Register OR bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_OVERRUN, NULL);
```

Arguments:

Table 5-564. SCI_GET_RX_OVERRUN ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_OVERRUN* ioctl command reads the SCI Status Register OR (Overrun Flag) bit.

Returns: *0* if the OR bit is cleared, non zero if the OR bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_OVERRUN* ioctl command is implemented as a macro.

Example 5-501. SCI_GET_RX_OVERRUN

```
tmp = ioctl(SCI_0, SCI_GET_RX_OVERRUN, NULL);
```

This code returns the SCI 0 Status Register, bit OR.

5.15.3.29 SCI_GET_RX_NOISE_ERROR - gets SCI Status Register NF bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_NOISE_ERROR,
          NULL);
```

Arguments:

Table 5-565. SCI_GET_RX_NOISE_ERROR ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_NOISE_ERROR* ioctl command reads the SCI Status Register NF (Noise Flag) bit.

Returns: *0* if the NF bit is cleared, non zero if the NF bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_NOISE_ERROR* ioctl command is implemented as a macro.

Example 5-502. SCI_GET_RX_NOISE_ERROR

```
tmp = ioctl(SCI_0, SCI_GET_RX_NOISE_ERROR, NULL);
```

This code returns the SCI 0 Status Register, bit NF.

5.15.3.30 SCI_GET_RX_FRAMING_ERROR - gets SCI Status Register FE bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_FRAMING_ERROR,
          NULL);
```

Arguments:

Table 5-566. SCI_GET_RX_FRAMING_ERROR ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_FRAMING_ERROR* ioctl command reads the SCI Status Register FE (Framing Error Flag) bit.

Returns: *0* if the FE bit is cleared, non zero if the FE bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_FRAMING_ERROR* ioctl command is implemented as a macro.

Example 5-503. SCI_GET_RX_FRAMING_ERROR

```
tmp = ioctl(SCI_0, SCI_GET_RX_FRAMING_ERROR, NULL);
```

This code returns the SCI 0 Status Register, bit FE.

5.15.3.31 SCI_GET_RX_PARITY_ERROR - gets SCI Status Register PF bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_PARITY_ERROR,
          NULL);
```

Arguments:

Table 5-567. SCI_GET_RX_PARITY_ERROR ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_PARITY_ERROR* ioctl command reads the SCI Status Register PF (Parity Error Flag) bit.

Returns: *0* if the PF bit is cleared, non zero if the PF bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_PARITY_ERROR* ioctl command is implemented as a macro.

Example 5-504. SCI_GET_RX_PARITY_ERROR

```
tmp = ioctl(SCI_0, SCI_GET_RX_PARITY_ERROR, NULL);
```

This code returns the SCI 0 Status Register, bit PF.

5.15.3.32 SCI_GET_RX_ACTIVE - gets SCI Status Register RAF bit

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_GET_RX_ACTIVE, NULL);
```

Arguments:

Table 5-568. SCI_GET_RX_ACTIVE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_ACTIVE* ioctl command reads the SCI Status Register RAF (Receiver Active Flag) bit.

Returns: *0* if the RAF bit is cleared, non zero if the RAF bit is set.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_GET_RX_ACTIVE* ioctl command is implemented as a macro.

Example 5-505. SCI_GET_RX_ACTIVE

```
tmp = ioctl(SCI_0, SCI_GET_RX_ACTIVE, NULL);
```

This code returns the SCI 0 Status Register, bit RAF.

5.15.3.33 SCI_LIN_MODE - enables or disables LIN slave operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_LIN_MODE, UWord16 param);
```

Arguments:

Table 5-569. SCI_LIN_MODE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable LIN slave operation or SCI_DISABLE to disable it.

Description: The *SCI_LIN_MODE* ioctl command enables or disables the LIN slave operation, modifying the LIN MODE bit in the SCI Control 2 Register. See the **56F8000 Peripheral Reference Manual** for more details.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F80xx.

Design/Implementation: The *SCI_LIN_MODE* ioctl command is implemented as a macro.

Example 5-506. SCI_LIN_MODE

```
ioctl( SCI_0, SCI_LIN_MODE, SCI_ENABLE);
```

This code enables the LIN slave operation on the SCI_0/SCI module.

5.15.3.34 SCI_QUEUED_MODE - enables or disables queued operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_QUEUED_MODE,
     UWord16 param);
```

Arguments:

Table 5-570. SCI_QUEUED_MODE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable queued operation or SCI_DISABLE to disable it.

Description: The *SCI_QUEUED_MODE* ioctl command enables or disables the receive and transmit FIFOs by modifying the FIFO_EN bit in the SCI Control 2 Register. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that for a queued operation, you still need to set-up the FIFO threshold levels by using the SCI_SET_RXFULL_CONDITION or SCI_SET_TXEMPTY_CONDITION ioctl commands.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SCI_QUEUED_MODE* ioctl command is implemented as a macro.

Example 5-507. SCI_QUEUED_MODE

```
ioctl(SCI, SCI_QUEUED_MODE, SCI_ENABLE);
```

This code enables queued mode of the SCI module.

5.15.3.35 SCI_SET_TXEMPTY_CONDITION - configures the transmit FIFO threshold in queued mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_SET_TXEMPTY_CONDITION,
     UWord16 param);
```

Arguments:

Table 5-571. SCI_SET_TXEMPTY_CONDITION ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use one of the predefined constants: SCI_TXEMPTY_WHEN_FIFO_EMPTY SCI_TXEMPTY_WHEN_1_OR_LESS_IN_FIFO SCI_TXEMPTY_WHEN_2_OR_LESS_IN_FIFO SCI_TXEMPTY_WHEN_3_OR_LESS_IN_FIFO

Description: The *SCI_SET_TXEMPTY_CONDITION* ioctl command configures the transmit FIFO threshold, which is the level at which the SCI signals to CPU that it requires new byte for transmission. The signaling is done a standard way using the Transmitter-Empty flag or interrupt. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that the queued mode must be enabled using the SCI_QUEUED_MODE ioctl command after this command, to achieve the desired “queued” functionality.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SCI_SET_TXEMPTY_CONDITION* ioctl command is implemented as a macro.

Example 5-508. SCI_SET_TXEMPTY_CONDITION

```
ioctl(SCI, SCI_SET_TXEMPTY_CONDITION,
     SCI_TXEMPTY_WHEN_FIFO_EMPTY);
```

This code sets the transmit FIFO threshold in a way that this FIFO must go completely empty before the SCI will signal TXEMPTY. In other words it means that when the TXEMPTY flag or interrupt is detected, up to four bytes (FIFO depth) can be written to the transmit register at once. See also the SCI_CAN_WRITE_DATA ioctl command.

5.15.3.36 SCI_SET_RXFULL_CONDITION - configures the receive FIFO threshold in queued mode

Call(s):

```
void ioctl(const int *pModuleBase, SCI_SET_RXFULL_CONDITION,
     UWord16 param);
```

Arguments:

Table 5-572. SCI_SET_RXFULL_CONDITION ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use one of the predefined constants: SCI_RXFULL_WHEN_1_OR_MORE_IN_FIFO SCI_RXFULL_WHEN_2_OR_MORE_IN_FIFO SCI_RXFULL_WHEN_3_OR_MORE_IN_FIFO SCI_RXFULL_WHEN_4_OR_MORE_IN_FIFO

Description: The *SCI_SET_RXFULL_CONDITION* ioctl command configures the receive FIFO threshold, which is the level at which the SCI signals to CPU that byte(s) were received and are available for reading-out of the SCI data register. The signaling is done a standard way using the Receiver-full flag or interrupt. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that the queued mode must be enabled using the SCI_QUEUED_MODE ioctl command after this command, to achieve the desired “queued” functionality.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SCI_SET_RXFULL_CONDITION* ioctl command is implemented as a macro.

Example 5-509. SCI_SET_RXFULL_CONDITION

```
ioctl(SCI, SCI_SET_RXFULL_CONDITION,
     SCI_RXFULL_WHEN_1_OR_MORE_IN_FIFO);
```

This code sets the receive FIFO threshold in a way that even a single character received causes the RXFULL flag to be set, just like in the non-queued mode. However, unlike the non-queued mode the receive FIFO accepts additional three characters (up to FIFO depth) before the receive-overrun occurs. See also the SCI_CAN_READ_DATA ioctl command.

5.15.3.37 SCI_CAN_READ_DATA - checks if data byte is available for reading

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_CAN_READ_DATA,
      NULL);
```

Arguments:

Table 5-573. SCI_CAN_READ_DATA ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_CAN_READ_DATA* ioctl command returns a non-zero value when there is at least one character received in the receive FIFO when operating in the queued mode. When a non-zero value is returned, the data may be read-out using the SCI_READ_DATA ioctl command.

Note that the queued mode must first be enabled using the SCI_QUEUED_MODE ioctl command to be able to use this command.

Returns: UWord16 for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SCI_CAN_READ_DATA* ioctl command is implemented as a macro.

Example 5-510. SCI_CAN_READ_DATA

```
UWord16 byte;

while(ioctl(SCI, SCI_CAN_READ_DATA, NULL))
{
      byte = ioctl(SCI, SCI_READ_DATA, NULL);
      ...

}
```

This code reads all bytes from the SCI receive FIFO.

5.15.3.38 SCI_CAN_WRITE_DATA - checks if a free slot is available in transmit FIFO

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI_CAN_WRITE_DATA,
      NULL);
```

Arguments:

Table 5-574. SCI_CAN_WRITE_DATA ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_CAN_WRITE_DATA* ioctl command returns a non-zero value when there is at least one free position left in the transmit FIFO when operating in the queued mode. When a non-zero value is returned, at least one data byte may be written to the SCI using the SCI_WRITE_DATA ioctl command.

Note that the queued mode must first be enabled using the SCI_QUEUED_MODE ioctl command to be able to use this command.

Returns: UWord16 for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SCI_CAN_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-511. SCI_CAN_WRITE_DATA

```
if(ioctl(SCI, SCI_CAN_WRITE_DATA, NULL))
{
      ioctl(SCI, SCI_WRITE_DATA, *pNextByte++);

}
```

This code writes one data byte into the transmit FIFO if possible.

5.15.3.39 read - read from SCI module

Call(s):

```
void read (const int *pModuleBase, NON_BLOCKING,
                                   int *address, int size);
void read (const int *pModuleBase, BLOCKING,
                                   int *address, int size);
int read (const int *pModuleBase, BUFFERED,
                                   int *address, int size);
```

Arguments:

Table 5-575. read function call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *read* function is implemented as macro and calls related *read* functions for non-blocking, blocking or buffered mode dependent on the *mode* parameter.

non-blocking mode: The *read* function in non-blocking mode initializes some internal status variables and pointers of the SCI driver, modifies the RIE (Receiver Full Interrupt Enable Bit) bit as well as the REIE (Receive Error Interrupt Enable Bit) bit in the SCI Control Register and returns immediately to the main program. All characters are sent within the interrupt service routine. Note: corresponding interrupt routines must be installed, see Table 5-576.

blocking mode: The *read* function in blocking mode uses a polling technique, interrupts are not used. The *read* function in blocking mode waits until all characters are read from the SCI module.

buffered mode: The *read* function in buffered mode copies the received characters from the internal circular buffer and frees the appropriate buffer space to enable reception of next incoming characters. Only available characters are fetched and returned. The function returns the number of characters read - which is the number equal or less than the *size* parameter

See e.g. Example 5-523 for more details.

Returns: In other than buffered mode, this call returns nothing. In buffered mode, this call returns the number of characters fetched from the receiver buffer.

Range Issues: Parameter *size* must greater than 1.

Special Issues: The *read* function in non-blocking mode is not reentrant, see Section 5.15.3.42 and Section .

The interrupt functions of the SCI driver must be installed to obtain proper functionality of the *read* function (See Table 5-576).

Table 5-576. interrupt routines of the SCI driver for the read function in non-blocking mode

void Sci*x*RxFullISR(void)	This function is dedicated to service SCI *x* Receiver Full Interrupt and must be installed at proper interrupt vector (See Example 5-523).
void Sci*x*RxErrorSR(void)	This function is dedicated to service SCI *x* Receiver Error Interrupt and must be installed at proper interrupt vector (See Example 5-523).

Note: Symbol *x* in the Table 5-576 should be replaced with the number of the SCI module e.g. *Sci0RxFullISR*.

non-blocking mode: None.

Design/Implementation: The *read* is implemented as a function call for all modes.

5.15.3.40 write - non-blocking or blocking write to SCI module

Call(s):

```
void write (const int *pModuleBase, NON_BLOCKING,
                                    int *address, int size);
void write (const int *pModuleBase, BLOCKING,
                                    int *address, int size);
int write (const int *pModuleBase, BUFFERED,
                                    int *address, int size);
```

Arguments:

Table 5-577. write function call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *write* function is implemented as a macro and calls related *write* functions for non-blocking and blocking mode, dependent on the *mode* parameter.

non-blocking mode: The *write* function in non-blocking mode initializes internal status variables and pointers of the SCI driver, modifies the TEIE (Transmitter Empty Interrupt Enable Bit) bit in the SCI Control Register and returns immediately to the main program. All characters are received within the interrupt service routines. Note: corresponding interrupt routines must be installed, see Table 5-578.

blocking mode: The *write* function in blocking mode uses a polling technique, interrupts are not used. The *write* function in blocking mode waits until all characters are written to the SCI.

buffered mode: The *write* function in buffered mode copies the acceptable number of characters to the internal circular buffer and returns immediately. The copied characters are then automatically transmitted on background. The function returns the number of characters accepted, which can be number equal or less than the *size* specified.

See e.g. Example 5-523 for more details.

Returns: In other than buffered mode, this call returns nothing. In buffered mode, this call returns the number of characters copied into the transmitter buffer.

Range Issues: Parameter *size* must greater than 1.

Special Issues: The *write* function in non-blocking mode is not reentrant, see Section 5.15.3.42 and Section .

The interrupt functions of the SCI driver must be installed to obtain proper functionality of the *write* function (See Table 5-578).

Table 5-578. interrupt routines of the SCI driver for the write function in non-blocking mode

void Sci*x*TxEmptyISR(void)	This function is dedicated to service SCI *x* Transmitter Empty Interrupt and must be installed at proper interrupt vector (See Example 5-523).

Note: Symbol *x* in the Table 5-576 should be replaced with the number of the SCI module e.g. *Sci0TxEmptyISR*.

non-blocking mode: None.

Design/Implementation: The *write* is implemented as a function call for all modes.

5.15.3.41 SCI_CLEAR_EXCEPTION - clears read/write function exception if exists

Call(s):

```
void ioctl(const int *pModuleBase, SCI_CLEAR_EXCEPTION, NULL);
```

Arguments:

Table 5-579. SCI_CLEAR_EXCEPTION ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_CLEAR_EXCEPTION* ioctl command clears a software error flags generated by *read*/*write* functions. The *SCI_CLEAR_EXCEPTION* ioctl command modifies some internal status variables of the SCI driver.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI_CLEAR_EXCEPTION* ioctl command is implemented as a macro.

Example 5-512. SCI_CLEAR_EXCEPTION

```
ioctl(SCI_0, SCI_CLEAR_EXCEPTION, NULL);
```

This code clears exceptions generated by read/write functions for the SCI 0 module.

5.15.3.42 SCI_GET_STATUS - read status word of the SCI software layer

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SCI0_GET_STATUS, NULL);
```

Arguments:

Table 5-580. SCI0_GET_STATUS ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI0_GET_STATUS* ioctl command reads the internal status word of the SCI driver.

Returns: The *SCI0_GET_STATUS* ioctl command returns the status word of the SCI software layer. The following status flags are defined:

SCI_STATUS_WRITE_INPROGRESS	NON_BLOCKING write operation is in progress
SCI_STATUS_READ_INPROGRESS	NON_BLOCKING read operation is in progress
SCI_STATUS_EXCEPTION_EXISTS	An error occurred during read of write operation.
SCI_STATUS_RX_BUFFER_LOW	The receive buffer-low condition exists (BUFFERED operation only)
SCI_STATUS_XOFF_PENDING	The priority XOFF character is to be sent as soon as possible (BUFFERED operation only)
SCI_STATUS_XON_PENDING	The priority XON character is to be sent as soon as possible (BUFFERED operation only)

The status word also contains the following software exception flags:

SCI_EXCEPTION_BUFFER_OVERFLOW	The receive buffer overflow occurred during NON_BLOCKING or BUFFERED operation. Some characters were lost.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SCI0_GET_STATUS* ioctl command is implemented as a macro.

Example 5-513. SCI0_GET_STATUS

```
tmp = ioctl(SCI_0, SCI0_GET_STATUS, NULL);
```

This code reads the status of the *read/write* function and stores it into variable *tmp*.

5.15.3.43 SCI_WRITE_CANCEL - cancels non-blocking write operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WRITE_CANCEL, NULL);
```

Arguments:

Table 5-581. SCI_WRITE_CANCEL ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_WRITE_CANCEL* ioctl command cancels any non-blocking write operation in case of non accomplishment.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BLOCKING / NON_BLOCKING high-level operation mode is selected only.

Design/Implementation: The *SCI_WRITE_CANCEL* ioctl command is implemented as a macro.

Example 5-514. SCI_WRITE_CANCEL

```
write(SCI_0, NON_BLOCKING, buffer, 10);
ioctl(SCI_0, SCI_WRITE_CANCEL, NULL);
```

This code cancels a non-blocking write operation into the SCI 0 module.

5.15.3.44 SCI_READ_CANCEL - cancels non-blocking read operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_READ_CANCEL, NULL);
```

Arguments:

Table 5-582. SCI_READ_CANCEL ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_READ_CANCEL* ioctl command cancels any non-blocking read operation in case of non accomplishment.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BLOCKING / NON_BLOCKING high-level operation mode is selected only.

Design/Implementation: The *SCI_READ_CANCEL* ioctl command is implemented as a macro.

Example 5-515. SCI_READ_CANCEL

```
read(SCI_0, NON_BLOCKING, buffer, 10);
ioctl(SCI_0, SCI_READ_CANCEL, NULL);
```

This code cancels a non-blocking read operation into the SCI 0 module.

5.15.3.45 SCI_BUFFERED_RX - activate buffered read operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_BUFFERED_RX,
     UWord16 param);
```

Arguments:

Table 5-583. SCI_BUFFERED_RX ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable or SCI_DISABLE to disable SCI buffered receive operation.

Description: The *SCI_BUFFERED_RX* ioctl enables or disables the buffered receive operation. When enabling, it initializes the receiver's circular buffer, initializes buffer pointers and status word and enables required interrupts.

After the BUFFERED operation is enabled, the read function can be called in BUFFERED mode.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_BUFFERED_RX* ioctl command is implemented as a function call.

Example 5-516. SCI_BUFFERED_RX

```
Word16 readChars;
ioctl(SCI_0, SCI_BUFFERED_RX, SCI_ENABLE);

...
readChars = read(SCI_0, BUFFERED, buffer, 10);
```

This code enables the buffered receive operation and reads up to 10 characters from the receivers circular buffer.

5.15.3.46 SCI_BUFFERED_TX - activate buffered write operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_BUFFERED_TX,
     UWord16 param);
```

Arguments:

Table 5-584. SCI_BUFFERED_TX ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.
param	in	Use SCI_ENABLE to enable or SCI_DISABLE to disable SCI buffered transmit operation.

Description: The *SCI_BUFFERED_TX* ioctl enables or disables the buffered transmit operation. When enabling, it initializes the transmitter's circular buffer, initializes buffer pointers and status word and enables required interrupts.

After the BUFFERED operation is enabled, the write function can be called in BUFFERED mode.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_BUFFERED_TX* ioctl command is implemented as a function call.

Example 5-517. SCI_BUFFERED_TX

```
ioctl(SCI_0, SCI_BUFFERED_TX, SCI_ENABLE);

...
write(SCI_0, BUFFERED, "Hello World", 11);
```

This code enables the buffered transmit operation and writes the "Hello World" string to the transmitter circular buffer.

5.15.3.47 SCI_GET_RX_CHARS_READY - get the number of characters available for buffered read operation

Call(s):

```
Word16 ioctl(const int *pModuleBase, SCI_GET_RX_CHARS_READY,
      NULL);
```

Arguments:

Table 5-585. SCI_GET_RX_CHARS_READY ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_CHARS_READY* ioctl returns the number of characters received and stored in the circular buffer. The characters are ready to be read in BUFFERED mode using the read call.

Returns: Number of valid characters in receiver buffer.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_GET_RX_CHARS_READY* ioctl command is implemented as a function call.

Example 5-518. SCI_GET_RX_CHARS_READY

```
if(ioctl(SCI_0, SCI_GET_RX_CHARS_READY, NULL) > 10)
      read(SCI_0, BUFFERED, buff, sizeof(buff));
```

This code checks whether any more than 10 characters are available in the receiver's buffer and reads them.

5.15.3.48 SCI_GET_RX_BUFFER_FREESPACE - get the free space in the receiver buffer

Call(s):

```
Word16 ioctl(const int *pModuleBase, SCI_GET_RX_BUFFER_FREESPACE,
      NULL);
```

Arguments:

Table 5-586. SCI_GET_RX_BUFFER_FREESPACE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_RX_BUFFER_FREESPACE* ioctl returns the number of characters the receiver in BUFFERED mode can still accept until the buffer overflow event occurs.

Returns: Number of free characters in receiver buffer.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_GET_RX_BUFFER_FREESPACE* ioctl command is implemented as a function call.

Example 5-519. SCI_GET_RX_BUFFER_FREESPACE

```
if(ioctl(SCI_0, SCI_GET_RX_BUFFER_FREESPACE, NULL) < 5)
{
      ...
}
```

This code checks whether the free space is below 5 characters.

5.15.3.49 SCI_GET_TX_BUFFER_FREESPACE - get the free space in the transmitter buffer

Call(s):

```
Word16 ioctl(const int *pModuleBase, SCI_GET_TX_BUFFER_FREESPACE,
     NULL);
```

Arguments:

Table 5-587. SCI_GET_TX_BUFFER_FREESPACE ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_GET_TX_BUFFER_FREESPACE* ioctl returns the number of characters the transmitter in BUFFERED mode can still accept using the buffered write call.

Returns: Number of free characters in transmitter buffer.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_GET_TX_BUFFER_FREESPACE* ioctl command is implemented as a function call.

Example 5-520. SCI_GET_TX_BUFFER_FREESPACE

```
if(ioctl(SCI_0, SCI_GET_TX_BUFFER_FREESPACE, NULL) >= 11)
     write(SCI_0, BUFFERED, "Hello World", 11);
```

This code sends the string "Hello World" after it assures the whole string will fit in the transmitter buffer.

5.15.3.50 SCI_SEND_XON - send priority XON character

Call(s):

```
Word16 ioctl(const int *pModuleBase, SCI_SEND_XON,
      NULL);
```

Arguments:

Table 5-588. SCI_SEND_XON ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_SEND_XON* ioctl call assures the XON character is transmitted as soon as possible. When the transmitter is idle, the XON character is put directly to the SCI data register. When the transmission is in progress, this command just sets the SCI_STATUS_XON_PENDING flag in the driver's status word and the XON character is then sent automatically after the current character is transmitted.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_SEND_XON* ioctl command is implemented as a macro.

Example 5-521. SCI_SEND_XON

```
ioctl(SCI_0, SCI_SEND_XON, NULL);
```

This code sends the XON character.

5.15.3.51 SCI_SEND_XOFF - send priority XOFF character

Call(s):

```
Word16 ioctl(const int *pModuleBase, SCI_SEND_XOFF,
     NULL);
```

Arguments:

Table 5-589. SCI_SEND_XOFF ioctl call arguments

*pModuleBase	in	The SCI module identifier, use SCI_0 or SCI_1. Note that SCI is equivalent to SCI_0 on devices with only one SCI module.

Description: The *SCI_SEND_XOFF* ioctl call assures the XOFF character is transmitted as soon as possible. When the transmitter is idle, the XOFF character is put directly to the SCI data register. When the transmission is in progress, this command just sets the SCI_STATUS_XOFF_PENDING flag in the driver's status word and the XOFF character is then sent automatically after the current character is transmitted.

Returns: None.

Range Issues: None.

Special Issues: This command can be applied when BUFFERED high-level operation mode is selected only.

Design/Implementation: The *SCI_SEND_XOFF* ioctl command is implemented as a macro.

Example 5-522. SCI_SEND_XOFF

```
ioctl(SCI_0, SCI_SEND_XOFF, NULL);
```

This code sends the XOFF character.

5.15.4 SCI Driver Application

The SCI driver application is designed for Freescale/Motorola EVMs and intended to illustrate the usage of this SCI driver.

The SCI driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM* directory and consists of the application project *sci_demo.mcp* and the source code for the application *main.c*.

Example 5-523. SCI Driver Sample Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.*************************************************************************
*
*  File generated by Graphical Configuration Tool Fri, 09/Feb/2007, 16:28:11
*
*************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
-------------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-------------------------------------------
```

```
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
.*/
#define SIM_GPS_INIT                    0x0000

/*.
    SEMI Configuration
-------------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                0x0005
#define SEMI_CSBAR1_INIT                0x0005
#define SEMI_CSBAR2_INIT                0x0005
#define SEMI_CSOR0_INIT                 0x1FC3
#define SEMI_CSOR1_INIT                 0x1BA3
#define SEMI_CSOR2_INIT                 0x1DA3

/*.
    INTC Configuration
-------------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0000
```

```
#define INT_VECTOR_ADDR_68                Sci0TxEmptyISR
#define INT_PRIORITY_LEVEL_68             INTC_LEVEL1
#define INT_VECTOR_ADDR_71                Sci0RxErrorISR
#define INT_PRIORITY_LEVEL_71             INTC_LEVEL1
#define INT_VECTOR_ADDR_72                Sci0RxFullISR
#define INT_PRIORITY_LEVEL_72             INTC_LEVEL1

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: ISA0 , PullUp: Enable ,
    Pin  9: Function: ISA1 , PullUp: Enable ,
    Pin 10: Function: ISA2 , PullUp: Enable ,
.*/
#define GPIO_C_DDR_INIT                   0x000F
#define GPIO_C_PER_INIT                   0x07F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  7: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_DDR_INIT                   0x00C0
#define GPIO_D_PER_INIT                   0x1F01

/*.
    SCI_0 Configuration
-----------------------------------------
    Baudrate: 58594 bps
    Enable Receiver: Disable
    Enable Transmitter: Disable
    Data word length: 8 bits
    Parity: None
    Polarity: True polarity
    Loop mode: Disable
    Function in Wait Mode: SCI module enabled in Wait Mode
.*/
#define SCI_0_SCICR_INIT                  0x0000
#define SCI_0_HIGHLEVEL_MODE              2
#define SCI_0_RX_BUFFER_OKLIMIT           0x000A
#define SCI_0_RX_BUFFER_LOWLIMIT          0x0003
#define SCI_0_RX_BUFFER_SIZE              0x14
#define SCI_0_TX_BUFFER_SIZE              0x14
```

```
#define SCI_0_RX_XONXOFF                0x01
#define SCI0CHAR                        char
#define SCI_0_RX_CHAR_HOOK              rx_char_hook


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-524. SCI Driver Sample Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application which shows how to read/write from/to SCI module
*
*              The application echoes back to PC any characters received on SCI.
*              When the '*' character is received, the application toggles the RED
*              LED and toggles also the character echoing on and off.
*
*              Connect the EVM to the PC Hyperterminal, select 9600kbps, XON/XOFF
*              and try to write. The EVM should echo the charaters back so they
*              should appear on the screen.
*
*              A short delay is forced after each SCI operation in the main loop,
*              so you may observe how XON/XOFF control characters are used to
*              pause the PC if it transmits too fast. Try to paste a large block
*              of text from a PC to the SCI and you should see it echoed back
*              correctly. Compare this result with the one after XON/XOFF is
*              disabled on the PC side.
*
*
* TARGET: MC56F8xxx device
*
*****************************************************************************/

#include "qs.h"

#include "gpio.h"
#include "occs.h"
#include "sci.h"
#include "intc.h"
#include "sys.h"
#include "cop.h"

/* board-specific LEDs */
#include "../board.h"

/* forward prototypes */
void device_init(void);

/* welcome message */
char buff[] = "Hello World";

/*
 *  The main
 */

int main(void)
{
    Word16 r;

    /* initialize SYS, COP and pins */
    device_init();

    /* initialize SCI1 @ 9600 kbps */
    ioctl(SCI_0, SCI_INIT, NULL);
```

```
    ioctl(SCI_0, SCI_SET_BAUDRATE, SCI_BAUD_9600);

    /* enable SCI1 peripheral */
    ioctl(SCI_0, SCI_TRANSMITTER, SCI_ENABLE);
    ioctl(SCI_0, SCI_RECEIVER, SCI_ENABLE);

    /* enable buffered read/write operations */
    ioctl(SCI_0, SCI_BUFFERED_RX, SCI_ENABLE);
    ioctl(SCI_0, SCI_BUFFERED_TX, SCI_ENABLE);

    /* initialize interrupt controller */
    ioctl(INTC, INTC_INIT, NULL);

    /* enable interrupts */
    archEnableInt();

    /* send welcome message */
    write(SCI_0, BUFFERED, buff, sizeof(buff)-1);

    while(1)
    {
        /* read any available characters */
        r = read(SCI_0, BUFFERED, buff, sizeof(buff));

        /* send them back */
        if(r)
            write(SCI_0, BUFFERED, buff, r);

        /* delay to show the buffering and XON/XOFF functionality */
        for(r=0; r<100; r++)
        {
            archDelay(0xffff);
            ioctl(COP, COP_CLEAR_COUNTER, NULL);
        }
    }
}


/*
 *  The callback function called for each character received (see appconfig.h)
 *
 *
 *  We return nonzero (true) when we want to "steal" the character which we
 *  do not want to be stored in a recive buffer.
 */

#pragma interrupt called

UWord16 rx_char_hook(UWord16 rxchar)
{
    static UWord16 rxEna = 0;

    /* toggle LED */
    ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_Y);

    /* for asterisk, togle LED and togle rx enable */
    if(rxchar == '*')
    {
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_R);
        rxEna = !rxEna;

        /* never echo asterisk */
        return 1;
    }

    /* throw character away if rxEna is non-zero */
```

```
    return rxEna;
}


/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.16 SPI Driver

This section describes the API for the MC56F83xx and MC56F80xx Serial Peripheral Interface (SPI). The functionality of the SPI module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual** and **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.16.1 Introduction

There are either one or two SPI modules on each 56F8xxx family member. The SPI module allows full-duplex, synchronous, serial communication between the controller and peripheral devices or other controllers. It features master and slave modes, double-buffered operation, transmission length 2-16bits, programmable frequency and various other options. Programmable clock options enables to adapt the SPI module to any SPI compatible device. This section describes the SPI driver software providing the lowest level software layer, interfacing hardware with software.

The MC56F802x/3x family newly adds a support for queued operation of SPI (receive and transmit FIFOs), enabling to lower the interrupt rate to CPU and/or to simplify software design in some scenarios. The SPI module with a queued capability is often referred as QSPI. However, the module identifier used in the low-level drivers of DSP56800E_Quick_Start and in *appconfig.h* initialization values remains SPI, SPI_0 or SPI_1 for backward compatibility.

5.16.2 Quick Reference

This section is intended to be a source of quick access information, while the details are discussed in Section 5.16.3.

Table 5-590. SPI Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F8037	MC56F83xx
SPI (SPI_BASE)	0xF100	0xF0C0	N/A	N/A	N/A
SPI0 (SPI0_BASE)	N/A	N/A	0xF220	0xF220	0xF2A0
SPI1 (SPI1_BASE)	N/A	N/A	N/A	0xF230	0xF2B0

5.16.2.1 API Definition

The following header files are needed in order to use the SPI device driver:

Required Header File(s):

```
#include "qs.h"
#include "spi.h"
```

The following information may be found in the header file *spi.h*.

Public Data Structure(s):

```
None
```

5.16.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static SPI module configuration by the driver initialization routine. Symbol *x* in the table should be replaced with the number of the SPI module e.g. SPI_0_SPDSCR_INIT. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-566 for more details.

Table 5-591. SPI Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
SPIxCHAR	type name char or UWord16	Specifies the type of single 'character' for buffer operations. The 'UWord16' is default and is backward compatible with HawkV1 implementation. The 'char' is the optimal choice for 56F800E family as it supports arrays or byte-wide elements.
SPI_*x*_SPDSCR_INIT	UWord16	Initial value of the SPI Data Size and Control Register.
SPI_*x*_SPSCR_INIT	UWord16	Initial value of the SPI Status and Control Register.
MC56F802x/3x only: SPI_*x*_FIFO_INIT	UWord16	Initial value of the SPI FIFO Control Register.
MC56F802x/3x only: SPI_*x*_DELAY_INIT	UWord16	Initial value of the SPI Word Delay Register.

In addition to standard configuration values described above, the DSP56800E_Quick_Start tool contains also SPI software interface which enables buffered operations in various configurations. To enable this built-in software layer, the following configuration items have to be specified in the *appconfig.h* file.

Table 5-592. SPI Operation Items for appconfig.h

SYMBOL	TYPE / VALUE	DESCRIPTION
INT_VECTOR_ADDR_yy yy = SPI0 Receiver Full Interrupt	Spi0RxFullISR	Name of the built in interrupt service routine implemented in *spi.c* file.
INT_VECTOR_ADDR_yy yy = SPI0 Transmitter Empty Interrupt	Spi0TxEmptyISR	Name of the built in interrupt service routine implemented in *spi.c* file.

Table 5-592. SPI Operation Items for appconfig.h (Continued)

SYMBOL	TYPE / VALUE	DESCRIPTION
INT_VECTOR_ADDR_yy yy = SPI1 Receiver Full Interrupt	Spi1RxFullISR	Name of the built in interrupt service routine implemented in *spi.c* file.
INT_VECTOR_ADDR_yy yy = SPI1 Transmitter Empty Interrupt	Spi1TxEmptyISR	Name of the built in interrupt service routine implemented in *spi.c* file.
INT_PRIORITY_LEVEL_yy yy = the four SPI interrupts as described above	INTC_LEVEL0 | INTC_LEVEL1 | INTC_LEVEL2	To enable the SPI interrupts, they must be assigned a interrupt priority level.

The built-in software layer is implemented in the *spibl.c* file. The file implements the interface using read and write operations in BLOCKING or NON_BLOCKING modes - this implementation is backward compatible with previous (HawkV1) implementations.

There are several *appconfig.h* configuration items for the built in software layer:

Table 5-593. Built-in Software Layer Configuration Items

SYMBOL	TYPE / VALUE	DESCRIPTION
SPI_*x*_RX_CHAR_HOOK	function name declared as: `UWord16 hook(UWord16 rxchar)`	The name of callback function which is called after each character received. Return nonzero to discard the received character.
SPI_*x*_RX_ERROR_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when any error is detected by the receiver.
SPI_*x*_TX_CHAR_HOOK	function name declared as: `UWord16 hook(UWord16 txchar)`	The name of callback function which is called after each character transmitted. Return nonzero when custom (asynchronous) character was inserted to the SPI transmit buffer.
SPI_*x*_RX_FINISHED_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when NON_BLOCKING read operation finishes,
SPI_*x*_TX_FINISHED_HOOK	function name declared as: `void hook(void)`	The name of callback function which is called when NON_BLOCKING write operation finishes,

5.16.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.

3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, void *param);
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
void read(const int *pModuleBase, const int mode, UWord16 *address,
     int size);
void write(const int *pModuleBase, const int mode, UWord16 *address,
     int size);
```

Description: The *ioctl* call "changes" SPI device modes or accesses the SPI register(s). The third ioctl parameter is either a value or a pointer, depending on the cmd type. The *read* or *write* calls call functions for block read or block write from or to the SPI module.

Arguments:

Table 5-594. SPI Driver Arguments - ioctl

pModuleBase	in	SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
Cmd	in	Commands found in *spi.h* which are used to modify the SPI module status and control registers. See Table 5-596.
param	in, inout	Used to pass the relevant data to ioctl function call.

Table 5-595. SPI Driver Arguments - read/write

pModuleBase	in	SPI module identifier. Use SPI_0 or SPI_1.
mode	in	Used to set the read/write functions mode. Use NON_BLOCKING or BLOCKING.
address	in, inout	Address of the allocated data buffer.
size	in	Number of words to be read (in case of read function call) or to be written (in case of write function call).

Item Separator Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-596. ioctl commands

cmd	param	Return	Description
SPI_INIT	NULL	None	Initialize selected SPI module by data from configuration file (appconfig.h).
SPI_DEVICE	SPI_ENABLE / SPI_DISABLE	None	Enable/disable SPI device.
SPI_SET_MODE	SPI_MASTER / SPI_SLAVE	None	Set SPI to MASTER (SPI_MASTER) or SLAVE (SPI_SLAVE) mode.
SPI_SET_ORDER	SPI_LSB_FIRST / SPI_MSB_FIRST	None	Selects SPI data shift bit ordering - MSB bit first (SPI_MSB_FIRST) or LSB bit first (SPI_LSB_FIRST).
SPI_SET_CLOCK_POLARITY	SPI_RISING_EDGE / SPI_FALLING_EDGE	None	Sets SPI clock edge which shifts out data - either falling edge (SPI_FALLING_EDGE) or rising edge (SPI_RISING_EDGE).
SPI_SET_CLOCK_PHASE	SPI_SCLK_EDGE / SPI_SS_EDGE	None	Determines the condition which starts data shift out of SLAVE device. Transmission is started by first SCLK edge (SPI_SCLK_EDGE) or falling edge on /SS pin (SPI_SS_EDGE).
SPI_SET_MODE_FAULT	SPI_ENABLE / SPI_DISABLE	None	Configures SPI device to use mode fault detection logic (SPI_ENABLE) or to disable mode fault checking (SPI_DISABLE).
SPI_SET_TX_DATA_SIZE	UWord16	None	Sets length of the data word in bits. Possible value is 2-16.
SPI_SET_WIRED_OR_MODE	SPI_NORMAL / SPI_OPEN_DRAIN	None	Sets wired-or mode of the SPI pins.
SPI_SET_BAUD_DIV	SPI_DIV2 / SPI_DIV8 / SPI_DIV16 / SPI_DIV32 / SPI_DIV64 / SPI_DIV128 / SPI_DIV256 new on **MC56F800x:** SPI_DIV512 / SPI_DIV1024 / SPI_DIV2048 / SPI_DIV4096	None	Sets SPI baud rate (clock) divisor.

Table 5-596. ioctl commands (Continued)

cmd	param	Return	Description
SPI_INT_ENABLE	SPI_TX_EMPTY | SPI_RX_FULL | SPI_ERROR	None	Enables the selected SPI interrupts.
SPI_INT_DISABLE	SPI_TX_EMPTY | SPI_RX_FULL | SPI_ERROR	None	Disables the selected SPI interrupts.
SPI_RX_FULL_INT SPI_TX_EMPTY_INT SPI_ERROR_INT	*SPI_ENABLE / SPI_DISABLE*	*None*	*Interrupt enable or disable for individual interrupt sources. The same operation possible with SPI_INT_ENABLE / DISABLE*
SPI_WRITE_DATA	UWord16	None	Writes data to SPI Data Transmit Register. When in MASTER mode it initiates transmission.
SPI_READ_DATA	NULL	UWord16	Reads SPI Data Receive Register. It contains previously received data.
SPI_GET_TX_EMPTY	NULL	UWord16	Reads Status and Control Register SPTE bit (“Transmitter Empty bit”).
SPI_GET_RX_FULL	NULL	UWord16	Reads Status and Control Register SPRF bit (“Receiver Full bit”).
SPI_GET_RX_OVERFLOW	NULL	UWord16	Reads Status and Control Register OVRF bit (“Overflow bit”).
SPI_GET_MODE_FAULT	NULL	UWord16	Reads Status and Control Register MODF bit (“Mode Fault bit”).
SPI_READ_CONTROL_REG	NULL	UWord16	Reads SPI Control Register.
SPI_WRITE_CONTROL_REG	UWord16	NULL	Writes to SPI Control Register.
SPI_GET_ERROR	NULL	UWord16	Tests Status and Control Register OVRF and MODF bits (test if any SPI error)
SPI_CLEAR_MODE_FAULT	NULL	None	Clears Status and Control Register MODF bit (“Mode Fault bit”).
only on MC56F802x/3x and MC56F800x: SPI_MULT_BAUD_DIV	SPI_MULT_DIV_1x / SPI_MULT_DIV_2x	None	Multiplies baud-rate divisor value by a given factor.

Table 5-596. ioctl commands (Continued)

cmd	param	Return	Description
only on MC56F802x/3x and MC56F800x: SPI_TEST_SS_INPUT	NULL	UWord16	Returns immediate state of the SS input pin (as zero / non-zero).
only on MC56F802x/3x and MC56F800x: SPI_SET_SS_MODE	SPI_SS_INPUT / SPI_SS_MANUAL_OUT / (SPI_SS_AUTO_ENVELOPE | SPI_SS_AUTO_STROBE)	None	Sets SS signal mode during SPI-Master operation.
only on MC56F802x/3x and MC56F800x: SPI_SET_SS_OUTPUT	UWord16 (zero / non-zero)	None	Sets the SS pin value when operating in SPI_SS_MANUAL_OUT mode.
only on MC56F802x/3x and MC56F800x: SPI_SET_SS_WIRED_OR_MODE	SPI_OPEN_DRAIN / SPI_NORMAL	None	Sets the SS pin mode.
only on MC56F802x/3x and MC56F800x: SPI_OVERRIDE_SS_INPUT	SPI_ENABLE / SPI_DISABLE	None	Overrides the internal SS input with a value of the SPMSTR bit.
only on MC56F802x/3x: SPI_QUEUED_MODE	SPI_ENABLE / SPI_DISABLE	None	Enables the SPI receive and transmit FIFOs.
only on MC56F802x/3x: SPI_SET_TXEMPTY_CONDITION	SPI_TXEMPTY_*condition* where *condition* is one of: WHEN_FIFO_EMPTY / WHEN_1_OR_LESS_IN_FIFO / WHEN_2_OR_LESS_IN_FIFO / WHEN_3_OR_LESS_IN_FIFO	None	Sets the transmit FIFO threshold when a new characters will be requested from the CPU (by signaling the TXEMPTY interrupt).
only on MC56F802x/3x: SPI_SET_RXFULL_CONDITION	SPI_RXFULL_*condition* where *condition* is one of: WHEN_1_OR_MORE_IN_FIFO WHEN_2_OR_MORE_IN_FIFO WHEN_3_OR_MORE_IN_FIFO WHEN_4_OR_MORE_IN_FIFO	None	Sets receive FIFO threshold when a new characters will be requested from the CPU (by signaling the TXEMPTY interrupt).
only on MC56F802x/3x: SPI_CAN_READ_DATA	NULL	UWord16	Returns non-zero if valid data are waiting in the receive FIFO.
only on MC56F802x/3x: SPI_CAN_WRITE_DATA	NULL	UWord16	Returns non-zero if there is a free space in the transmit FIFO.
High-level software layer commands:			
SPI_CLEAR_EXCEPTION	NULL	None	Clears read/write functions exception if exists
SPI_GET_STATUS	NULL	UWord16	Get read/write functions SPI operation status

Table 5-596. ioctl commands (Continued)

cmd	param	Return	Description
SPI_WRITE_CANCEL	NULL	None	Cancels non-blocking write operation
SPI_READ_CANCEL	NULL	None	Cancels non-blocking read operation

Table 5-597. read/write modes

Mode	Return	Description
NON_BLOCKING	None	Calls read/write function in non-blocking mode. Internal functions of the SPI driver are used to read/write data. See Section 5.16.3.38 and Section 5.16.3.39.
BLOCKING	None	Calls read/write function in blocking mode. Internal functions of the SPI driver are used. See Section 5.16.3.38 and Section 5.16.3.39.

5.16.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.16.3.1 SPI_INIT - initialize the SPI module

Call(s):

```
void ioctl(const int *pModuleBase, SPI_INIT, NULL);
```

Arguments:

Table 5-598. SPI_INIT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_INIT* ioctl command executes the initialization function, which initializes all configured SPI registers by the values defined in the *appconfig.h* file. It is intended for static configuration of the SPI module after power-up. This command should be issued before using the SPI module.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_INIT* ioctl command is implemented as a function call. Only necessary SPI registers, which are defined in *appconfig.h*, are initialized. Execution time depends on the quantity of defined registers.

Example 5-525. SPI_INIT

```
ioctl(SPI_0, SPI_INIT, NULL);
```

This code initializes the SPI module with values defined in *appconfig.h*. The *appconfig.h* file can be edited manually or generated by the Graphical Configuration Tool.

5.16.3.2 SPI_DEVICE - enable/disable SPI device

Call(s):

```
void ioctl(const int *pModuleBase, COP_DEVICE, UWord16 param);
```

Arguments:

Table 5-599. SPI_DEVICE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use SPI_ENABLE to enable or SPI_DISABLE to disable SPI interface.

Description: The *SPI_DEVICE* ioctl command enables (SPI_ENABLE) / disables (SPI_DISABLE) the SPI interface. It sets the SPI Status and Control Register, bit SPE. Before using the SPI interface, the SPI device should be configured (in disabled state) and then enabled. This command writes directly in the SPI Status and Control Register SPE ("SPI Enable") bit.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_DEVICE* command is implemented as a macro.

Example 5-526. SPI_DEVICE

```
/* configure desired SPI parameters */
/* ... */
/* enable it */
ioctl(SPI_0, SPI_DEVICE, SPI_ENABLE);
```

This code enables the SPI interface. Before enabling, the SPI module should be properly configured.

5.16.3.3 SPI_SET_MODE - set master or slave mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_MODE, UWord16 param);
```

Arguments:

Table 5-600. SPI_SET_MODE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired mode. Use SPI_MASTER for Master or SPI_SLAVE for Slave mode.

Description: The *SPI_SET_MODE* ioctl command sets the operation mode of the SPI interface. For Master mode use SPI_MASTER and for Slave mode use SPI_SLAVE. In Master mode, the SPI can initiate transmission and generate a clock signal, as opposed to Slave mode. This command writes directly to the SPI Status and Control Register, bit SPMSTR ("Master Slave").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_SET_MODE* ioctl command is implemented as a macro.

Example 5-527. SPI_SET_MODE

```
ioctl(SPI_0, SPI_SET_MODE, SPI_MASTER);
```

This code sets the SPI to Master mode.

5.16.3.4 SPI_SET_ORDER - set ordering of bits in data word

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_ORDER, UWord16 param);
```

Arguments:

Table 5-601. SPI_SET_ORDER ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired order. Use SPI_LSB_FIRST for LSB bit first or SPI_MSB_FIRST for MSB bit first shifted out.

Description: The *SPI_SET_ORDER* ioctl command sets the order of bits in the data word transfer. For the LSB bit shifted out first, use SPI_LSB_FIRST or for the MSB bit first, use SPI_MSB_FIRST. This command writes directly to the SPI Status and Control Register, bit DSO ("Data Shift Order").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_SET_ORDER* ioctl command is implemented as a macro.

Example 5-528. SPI_SET_ORDER

```
ioctl(SPI_0, SPI_SET_ORDER, SPI_MSB_FIRST);
```

This code sets the SPI to shift out the MSB bit first.

5.16.3.5 SPI_SET_CLOCK_POLARITY - set active edge of the SPI clock signal

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_CLOCK_POLARITY,
           UWord16 param);
```

Arguments:

Table 5-602. SPI_SET_CLOCK_POLARITY ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired active edge. For shift out data on rising clock edge use SPI_RISING_EDGE or for falling edge use SPI_FALLING_EDGE.

Description: The *SPI_SET_CLOCK_POLARITY* ioctl command sets the active edge of the clock signal (SCLK), on which the data is shifted out. Use SPI_RISING_EDGE for rising edge or SPI_FALLING_EDGE for falling edge. This command writes directly to the SPI Status and Control Register, bit CPOL ("Clock Polarity").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_SET_CLOCK_POLARITY* ioctl command is implemented as a macro.

Example 5-529. SPI_SET_CLOCK_POLARITY

```
ioctl(SPI_0, SPI_SET_CLOCK_POLARITY, SPI_FALLING_EDGE);
```

This code sets the SPI to shift out data on the falling edge of the SCLK signal.

5.16.3.6 SPI_SET_CLOCK_PHASE - control the condition which starts data shift out for SLAVE device

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_CLOCK_PHASE,
           UWord16 param);
```

Arguments:

Table 5-603. SPI_SET_CLOCK_PHASE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired active edge. Use SPI_SCLK_EDGE to start transmission by first SCLK edge or SPI_SS_EDGE by falling edge on $\overline{SS}$ pin.

Description: The *SPI_SET_CLOCK_PHASE* ioctl command determines which signal starts the transmission of the first data bit from the SLAVE output. When SPI_SCLK_EDGE is used, the transmission is started by the first SCLK edge, for SPI_SS_EDGE the falling edge of the $\overline{SS}$ signal starts transmission. This command writes directly in the SPI Status and Control Register, bit CPHA (“Clock Phase”).

Returns: None.

Range Issues: None.

Special Issues: None

Design/Implementation: The SPI_SET_CLOCK_PHASE ioctl command is implemented as a macro.

Example 5-530. SPI_SET_CLOCK_PHASE

```
ioctl(SPI_0, SPI_SET_CLOCK_PHASE, SPI_SS_EDGE);
```

This code sets the SPI to start shift data on the falling edge of the $\overline{SS}$ signal.

5.16.3.7 SPI_SET_MODE_FAULT - enable/disable mode fault detection

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_MODE_FAULT,
           UWord16 param);
```

Arguments:

Table 5-604. SPI_SET_MODE_FAULT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired mode. For mode fault detection enabled use SPI_ENABLE or to disable it SPI_DISABLE.

Description: The *SPI_SET_MODE_FAULT* ioctl command enables/disables the mode fault detection. If enabled, this detection sets the MODF flag bit if the /SS pin goes high during transmission in Slave mode or when /SS goes low at any time in Master mode. Use SPI_ENABLE for error detection or SPI_DISABLE to disable it. This command writes directly in the SPI Status and Control Register, bit MODFEN (“Mode Fault Enable”).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_SET_MODE_FAULT* ioctl command is implemented as a macro.

Example 5-531. SPI_SET_MODE_FAULT

```
ioctl(SPI_0, SPI_SET_MODE_FAULT, SPI_ENABLE);
```

This code enables the SPI mode fault error detection.

5.16.3.8 SPI_SET_TX_DATA_SIZE - set data word length

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_TX_DATA_SIZE,
           UWord16 param);
```

Arguments:

Table 5-605. SPI_SET_TX_DATA_SIZE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select data word length in bits. Range 2-16.

Description: The *SPI_SET_TX_DATA_SIZE* ioctl command sets the number of bits in one transmission word. Allowed range is 2-16 bits. This command writes directly in the SPI Data Size Register. The param value is decremented before writing to the register.

Returns: None.

Range Issues: param: 2-16

Special Issues: None.

Design/Implementation: The *SPI_SET_TX_DATA_SIZE* ioctl command is implemented as a macro.

Example 5-532. SPI_SET_TX_DATA_SIZE

```
ioctl(SPI_0, SPI_SET_TX_DATA_SIZE, 8);
```

This code sets the SPI transmission word length to 8 bit.

5.16.3.9 SPI_SET_WIRED_OR_MODE - set wired-or mode of the SPI pins

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_WIRED_OR_MODE,
           UWord16 param);
```

Arguments:

Table 5-606. SPI_SET_WIRED_OR_MODE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use SPI_NORMAL to disable pull-ups or SPI_OPEN_DRAIN to enable pull-ups.

Description: The *SPI_SET_WIRED_OR_MODE* ioctl command disables (SPI_NORMAL) or/ enables (SPI_OPEN_DRAIN) pull-ups for the SPI pins. This command writes directly in the SPI Data Size and Control Register WOM ("Wired-OR Mode") bit.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_SET_WIRED_OR_MODE command is implemented as a macro.

Example 5-533. SPI_SET_WIRED_OR_MODE

```
ioctl(SPI_0, SPI_SET_WIRED_OR_MODE, SPI_OPEN_DRAIN);
```

This code enables pull-ups for the SPI pins.

5.16.3.10 SPI_SET_BAUD_DIV - set baud rate (clock) divisor

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_BAUD_DIV,
           UWord16 param);
```

Arguments:

Table 5-607. SPI_SET_BAUD_DIV ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired SPI clock divisor. Use SPI_DIV2 for IPbus/2, SPI_DIV8 for IPbus/8, SPI_DIV16 for IPbus/16, SPI_DIV32 for IPbus/32, SPI_DIV64 for IPbus/64, SPI_DIV128 for IPbus/128 or SPI_DIV256 for IPbus/256. new on **MC56F800x:** Use SPI_DIV512 for IPbus/512, SPI_DIV1024 for IPbus/1024, SPI_DIV2048 for IPbus/2048, SPI_DIV4096 for IPbus/4096.

Description: The *SPI_SET_BAUD_DIV* ioctl command selects the baud rate (clock) divisor. This command writes directly in the SPI Status and Control Register, bits SPR1:SPR0 (“Baud Rate Select”).

This command for MC56F800x devices writes into the SPI Status and Control Register, bits SPR2:SPR0 (“Baud Rate Select”) and the SPI Data Size and Control Register, bit SPR3 (“SPI Baud Rate Select”)

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_SET_BAUD_DIV* ioctl command is implemented as a macro.

Example 5-534. SPI_SET_BAUD_DIV

```
ioctl(SPI_0, SPI_SET_BAUD_DIV, SPI_DIV8);
```

This code sets the SPI clock to IPbus frequency divided by 8 (usually 5MHz for 40MHz IPbus clock).

5.16.3.11 SPI_RX_FULL_INT - enable/disable SPI receive interrupt

Call(s):

```
void ioctl(const int *pModuleBase, SPI_RX_FULL_INT,
           UWord16 param);
```

Arguments:

Table 5-608. SPI_RX_FULL_INT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_ENABLE to enable SPI Receiver interrupt or SPI_DISABLE to disable it.

Description: The *SPI_RX_FULL_INT* ioctl command is used to enable/disable the SPI Receiver interrupt. It sets the value of the SPI Status and Control Register 1, bit SPRIE ("Receiver Interrupt Enable").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_RX_FULL_INT* command is implemented as a macro.

Example 5-535. SPI_RX_FULL_INT

```
/* enable SPI receive interrupt */
ioctl(SPI_0, SPI_RX_FULL_INT, SPI_ENABLE);
/* ... */
/* disable SPI receive interrupt */
/ioctl(SPI_0, SPI_RX_FULL_INT, SPI_DISABLE);
```

This code switches on/off the SPI receiver interrupt.

5.16.3.12 SPI_TX_EMPTY_INT - enable/disable SPI transmit interrupt

Call(s):

```
void ioctl(const int *pModuleBase, SPI_TX_EMPTY_INT,
           UWord16 param);
```

Arguments:

Table 5-609. SPI_TX_EMPTY_INT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_ENABLE to enable SPI Transmitter interrupt or SPI_DISABLE to disable it.

Description: The *SPI_TX_EMPTY_INT* ioctl command is used to enable/disable the SPI Receiver interrupt. It sets the value of the SPI Status and Control Register 1, bit SPTIE ("Transmit Interrupt Enable").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_TX_EMPTY_INT* command is implemented as a macro.

Example 5-536. SPI_TX_EMPTY_INT

```
/* enable SPI transmit interrupt */
ioctl(SPI_0, SPI_TX_EMPTY_INT, SPI_ENABLE);
/* ... */
/* disable SPI transmit interrupt */
/ioctl(SPI_0, SPI_TX_EMPTY_INT, SPI_DISABLE);
```

This code switches on/off the SPI transmit interrupt.

5.16.3.13 SPI_ERROR_INT - enable/disable SPI error interrupt

Call(s):

```
void ioctl(const int *pModuleBase, SPI_ERROR_INT, UWord16 param);
```

Arguments:

Table 5-610. SPI_ERROR_INT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_ENABLE to enable SPI error interrupt (caused by MODF or OVRF flags) or SPI_DISABLE to disable it.

Description: The *SPI_ERROR_INT* ioctl command is used to enable/disable the SPI Error interrupt. It sets the value of the SPI Status and Control Register 1, bit ERRIE ("Error Interrupt Enable").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_ERROR_INT* command is implemented as a macro.

Example 5-537. SPI_ERROR_INT

```
/* enable SPI error interrupt */
ioctl(SPI_0, SPI_ERROR_INT, SPI_ENABLE);
/* ... */
/* disable SPI error interrupt */
/ioctl(SPI_0, SPI_ERROR_INT, SPI_DISABLE);
```

This code switches on/off the SPI error interrupt.

5.16.3.14 SPI_INT_ENABLE - enable selected SPI interrupts

Call(s):

```
void ioctl(const int *pModuleBase, SPI_INT_ENABLE, UWord16 param);
```

Arguments:

Table 5-611. SPI_INT_ENABLE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_TX_EMPTY for SPI Transmitter Empty interrupt, SPI_RX_FULL for SPI Receiver Full interrupt or SPI_ERROR for SPI Error interrupt.

Description: The *SPI_INT_ENABLE* ioctl command is used to enable selected SPI interrupts. It sets the value of the SPI Status and Control Register 1, bit SPRIE (“Receiver Interrupt Enable”), bit SPTIE (“Transmit Interrupt Enable”) and bit ERRIE (“Error Interrupt Enable”).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_INT_ENABLE command is implemented as a macro.

Example 5-538. SPI_INT_ENABLE

```
/* enable SPI interrupts */
ioctl(SPI_0, SPI_INT_ENABLE, SPI_TX_EMPTY | SPI_RX_FULL);
```

This code enables the SPI Receiver Full and Transmitter Empty interrupt.

5.16.3.15 SPI_INT_DISABLE - disable selected SPI interrupts

Call(s):

```
void ioctl(const int *pModuleBase, SPI_INT_DISABLE,
           UWord16 param);
```

Arguments:

Table 5-612. SPI_INT_DISABLE ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_TX_EMPTY for SPI Transmitter Empty interrupt, SPI_RX_FULL for SPI Receiver Full interrupt or SPI_ERROR for SPI Error interrupt.

Description: The *SPI_INT_ENABLE* ioctl command is used to disable selected SPI interrupts. It clears the value of the SPI Status and Control Register 1, bit SPRIE ("Receiver Interrupt Enable"), bit SPTIE ("Transmit Interrupt Enable") and bit ERRIE ("Error Interrupt Enable").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_INT_DISABLE command is implemented as a macro.

Example 5-539. SPI_INT_DISABLE

```
/* enable SPI interrupts */
ioctl(SPI_0, SPI_INT_DISABLE, SPI_TX_EMPTY | SPI_RX_FULL);
```

This code disables the SPI Receiver Full and Transmitter Empty interrupt.

5.16.3.16 SPI_INT_SELECT - disable all SPI interrupts and enables selected

Call(s):

```
void ioctl(const int *pModuleBase, SPI_INT_SELECT, UWord16 param);
```

Arguments:

Table 5-613. SPI_INT_SELECT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Parameter to select desired action. Use SPI_TX_EMPTY for SPI Transmitter Empty interrupt, SPI_RX_FULL for SPI Receiver Full interrupt or SPI_ERROR for SPI Error interrupt.

Description: The *SPI_INT_ENABLE* ioctl command is used to enable selected SPI interrupts and disable the other ones. It sets the value of the SPI Status and Control Register 1, bit SPRIE ("Receiver Interrupt Enable"), bit SPTIE ("Transmit Interrupt Enable") and bit ERRIE ("Error Interrupt Enable").

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_INT_SELECT command is implemented as a macro.

Example 5-540. SPI_INT_SELECT

```
/* enable SPI interrupts */
ioctl(SPI_0, SPI_INT_ENABLE,
          SPI_TX_EMPTY | SPI_RX_FULL | SPI_ERROR);
/* ... */
/* disable SPI error interrupt */
/ioctl(SPI_0, SPI_INT_SELECT, SPI_RX_FULL);
```

After execution of this code SPI Receiver Full interrupt remains enabled, SPI Transmitter Empty and SPI Receiver error will be disabled.

5.16.3.17 SPI_WRITE_DATA - write Data Transmit Register

Call(s):

```
void ioctl(const int *pModuleBase, SPI_WRITE_DATA, UWord16 param);
```

Arguments:

Table 5-614. SPI_WRITE_DATA ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Data word to transmit. Range according to SPI Data Size Register setting (SPI_SET_TX_DATA_SIZE).

Description: The *SPI_WRITE_DATA* ioctl command writes data to the SPI Data Transmit Register. When in MASTER mode, it initiates data transmission.

Returns: None.

Range Issues: According to the setting of *SPI_SET_TX_DATA_SIZE*.

Special Issues: None.

Design/Implementation: The *SPI_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-541. SPI_WRITE_DATA

```
/* SPI is properly configured and enabled */
/* write data to SPI bus (if in Master mode) */
ioctl(SPI_0, SPI_WRITE_DATA,0x58A4);
```

This code puts 1 data word on the SPI bus.

5.16.3.18 SPI_READ_DATA - read Data Receive Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_READ_DATA, NULL);
```

Arguments:

Table 5-615. SPI_READ_DATA ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_READ_DATA* ioctl command reads data from the SPI Data Receive Register. It contains the previously received data word.

Returns: Received data word.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_READ_DATA* ioctl command is implemented as a macro.

Example 5-542. SPI_READ_DATA

```
UWord16 D1;
/* SPI is properly configured and enabled */
/* read previously received data */
D1 = ioctl(SPI_0, SPI_READ_DATA, NULL);
```

This code reads the data word received from the SPI bus.

5.16.3.19 SPI_GET_TX_EMPTY - read transmitter empty interrupt flag

Call(s):

```
bool ioctl(const int *pModuleBase, SPI_GET_TX_EMPTY, NULL);
```

Arguments:

Table 5-616. SPI_GET_TX_EMPTY ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_TX_EMPTY* ioctl command reads the SPI Status and Control Register, bit SPTE ("Transmitter Empty"). This bit is set when the data transmit register transfers data into the shift register. For a more detailed description of the SPTE bit and associated timing, see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

Returns: True if SPTE is set, false otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_TX_EMPTY* command is implemented as a macro.

Example 5-543. SPI_GET_TX_EMPTY

```
/* test status and control register SPTE bit */
if (ioctl(SPI_0, SPI_GET_TX_EMPTY, NULL))
{
/* Data Transmit Register ready to receive new data word */
ioctl(SPI_0, SPI_WRITE_DATA, 0x0057);
}
```

This code tests the SPTE bit in the Status and Control Register and writes a new word into the Data Transmit Register, if empty.

5.16.3.20 SPI_GET_RX_FULL - read receiver full interrupt flag

Call(s):

```
bool ioctl(const int *pModuleBase, SPI_GET_RX_FULL, NULL);
```

Arguments:

Table 5-617. SPI_GET_RX_FULL ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_RX_FULL* ioctl command reads the SPI Status and Control Register, bit SPRF ("Receiver Full"). This bit is set when data is transferred from the shift register to the Receive Data Register. For a more detailed description of the SPRF bit and associated timing, see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

Returns: True if SPRF is set, false otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_RX_FULL* command is implemented as a macro.

Example 5-544. SPI_GET_RX_FULL

```
UWord16 D1;
/* test status and control register SPRF bit */
if (ioctl(SPI_0, SPI_GET_RX_FULL, NULL))
{
/* new data in Data Receive Register ready */
D1 = ioctl(SPI_0, SPI_READ_DATA, NULL);
}
```

This code tests the status and control register, bit SPRF and reads a new word from the Data Receive Register, if any.

5.16.3.21 SPI_GET_RX_OVERFLOW - read receiver overflow flag

Call(s):

```
bool ioctl(const int *pModuleBase, SPI_GET_RX_FULL, NULL);
```

Arguments:

Table 5-618. SPI_GET_RX_OVERFLOW ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_RX_OVERFLOW* ioctl command reads the SPI Status and Control Register, bit OVRF ("Overflow"). This bit is set when new data is received and previously received data have not been read from the SPI Data Receive Register. For a more detailed description of the OVRF bit and associated timing, see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

Returns: True if OVRF is set, false otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_RX_OVERFLOW* command is implemented as a macro.

Example 5-545. SPI_GET_RX_OVERFLOW

```
UWord16 D1;
/* test status and control register OVRF bit */
if (ioctl(SPI_0, SPI_GET_RX_OVERFLOW, NULL))
{
/* new data received and previous data have not been fetched yet,
new data is lost and it is only possible to read previous data */
/* first overflow flag should be cleared and then read data */
ioctl(SPI_0, SPI_CLEAR_RX_OVERFLOW, NULL);
D1 = ioctl(SPI_0, SPI_READ_RX_REG, NULL);
}
```

This code tests the Status and Control Register, bit OVRF. If the bit is set, it reads the previous data word from the Data Receive Register and clears bit OVRF.

5.16.3.22 SPI_READ_CONTROL_REG - read SPI Status and Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_READ_CONTROL_REG,
              NULL);
```

Arguments:

Table 5-619. SPI_READ_CONTROL_REG ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_READ_CONTROL_REG* ioctl command reads the SPI Status and Control Register

Returns: Content of the SPI Status and Control Register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_READ_CONTROL_REG command is implemented as a macro.

Example 5-546. SPI_READ_CONTROL_REG

```
UWord16 reg;

/* read SPI0 Status and Control Register */
reg = ioctl(SPI_0, SPI_READ_CONTROL_REG, NULL);
```

This code reads the SPI Status and Control Register and stores its value in the *reg* variable.

5.16.3.23 SPI_WRITE_CONTROL_REG - write SPI Status and Control Register

Call(s):

```
void ioctl(const int *pModuleBase, SPI_WRITE_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-620. SPI_WRITE_CONTROL_REG ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	UWord16

Description: The *SPI_WRITE_CONTROL_REG* ioctl command writes in the SPI Status and Control Register

Returns: None

Range Issues: None.

Special Issues: None.

Design/Implementation: The SPI_WRITE_CONTROL_REG command is implemented as a macro.

Example 5-547. SPI_WRITE_CONTROL_REG

```
/* write SPI0 Status and Control Register */
ioctl(SPI_0, SPI_WRITE_CONTROL_REG, 0x0020);
```

This code writes 0x0020 in the SPI Status and Control Register.

5.16.3.24 SPI_GET_MODE_FAULT - read mode fault flag

Call(s):

```
bool ioctl(const int *pModuleBase, SPI_GET_MODE_FAULT, NULL);
```

Arguments:

Table 5-621. SPI_GET_MODE_FAULT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_MODE_FAULT* ioctl command reads the SPI Status and Control Register, bit MODF ("Mode Fault"). This bit is set if the /SS pin goes high during transmission in Slave mode and if /SS goes low at any time in Master mode. This bit can only be set if enabled by the SPI_SET_MODE_FAULT command. For a more detailed description of the bit MODF and associated timing, see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

Returns: True if MODF is set, false otherwise.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_MODE_FAULT* command is implemented as a macro.

Example 5-548. SPI_GET_GET_MODE_FAULT

```
/* mode fault checking is enabled */
/* test status and control register MODF bit */
if (ioctl(SPI_0, SPI_GET_MODE_FAULT, NULL))
{
/* error in /SS level occurred */
ioctl(SPI_0, SPI_CLEAR_MODE_FAULT, NULL);
}
```

This code tests the Status and Control Register, bit MODF. This bit is cleared, if it is set.

5.16.3.25 SPI_GET_ERROR - test if any SPI error occurred

Call(s):

```
bool ioctl(const int *pModuleBase, SPI_GET_ERROR, NULL);
```

Arguments:

Table 5-622. SPI_GET_ERROR ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_ERROR* ioctl command tests the SPI Status and Control Register, bit OVRF ("Receiver Full") and bit MODF ("Mode Fault"). If any of these bits is set, this command returns true. This command is useful to test receive errors in receive interrupt subroutines because there is only 1 interrupt vector for both receive and error interrupt sources.

Returns: True if OVRF or MODF is set. When both are cleared, it returns false.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_ERROR* command is implemented as a macro.

Example 5-549. SPI_GET_ERROR

```
/* test status and control register MODF and OVRF bit */
if (ioctl(SPI_0, SPI_GET_ERROR, NULL))
{
/* error occurred */
/* to distinguish between MODF and OVRF SPI_GET_MODE_FAULT or
SPI_GET_RXD_OVERFLOW commands can be used */
if (ioctl(SPI_0, SPI_GET_RXD_OVERFLOW, NULL))
{
/* receive overflow */
}
else
{
/* mode fault error */
ioctl(SPI_0, SPI_CLEAR_MODE_FAULT, NULL);
}
}
```

This code tests Status and Control Register, bits OVRF and MODF and shows possible error states.

5.16.3.26 SPI_CLEAR_MODE_FAULT - clear mode fault error flag

Call(s):

```
void ioctl(const int *pModuleBase, SPI_CLEAR_MODE_FAULT, NULL);
```

Arguments:

Table 5-623. SPI_CLEAR_MODE_FAULT ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_CLEAR_MODE_FAULT* ioctl command clears the SPI Status and Control Register, bit MODF ("Mode Fault"). The MODF bit is cleared by writing "1" to this bit.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_CLEAR_MODE_FAULT* command is implemented as a macro.

Example 5-550. SPI_CLEAR_MODE_FAULT

```
/* clear status and control register MODF bit */
ioctl(SPI_0, SPI_CLEAR_MODE_FAULT, NULL))
```

This code clears the Status and Control Register, bit MODF.

5.16.3.27 SPI_MULT_BAUD_DIV - select additional baud-rate divisor factor 1x or 2x

Call(s):

```
void ioctl(const int *pModuleBase, SPI_MULT_BAUD_DIV,
     UWord16 param);
```

Arguments:

Table 5-624. SPI_MULT_BAUD_DIV ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_MULT_DIV_1x ... use standard divisor value SPI_MULT_DIV_2x ... multiply standard divisor value by factor 2

Description: The *SPI_MULT_BAUD_DIV* ioctl command enables or disables the additional 2x baud-rate divisor. This additional divisor is applied together with the standard one controlled by the SPI_SET_BAUD_DIV ioctl command.

This command directly sets or clears the BD2X bit in the SPI Data Size and Control Register (SPDSCR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_MULT_BAUD_DIV* ioctl command is implemented as a macro.

Example 5-551. SPI_MULT_BAUD_DIV

```
ioctl(SPI, SPI_SET_BAUD_DIV, SPI_DIV256);
ioctl(SPI, SPI_MULT_BAUD_DIV, SPI_ENABLE);
```

This code sets total baud-rate divisor value of 512.

5.16.3.28 SPI_TEST_SS_INPUT - test immediate state of the SS input pin

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_TEST_SS_INPUT, NULL);
```

Arguments:

Table 5-625. SPI_TEST_SS_INPUT ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_TEST_SS_INPUT* ioctl command returns immediate state of the slave select (SS) input pin regardless of which mode the SPI operates in. This bit is returned as a UWord16 value, which may only be tested in boolean-like (zero or non-zero) conditions. See the **56F802x/3x Peripheral Reference Manual** for more details.

This command returns the value of the SS_IN bit of the SPI Data Size and Control Register (SPDSCR).

Returns: UWord16 for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_TEST_SS_INPUT* ioctl command is implemented as a macro.

Example 5-552. SPI_TEST_SS_INPUT

```
if(ioctl(SPI, SPI_TEST_SS_INPUT, NULL))
{
      ...
}
```

This code tests the immediate state of the SS input pin.

5.16.3.29 SPI_SET_SS_MODE - set the SS signal mode in SPI Master mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_SS_MODE,
     UWord16 param);
```

Arguments:

Table 5-626. SPI_SET_SS_MODE ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_SS_INPUT ... set SS as input SPI_SS_MANUAL_OUT ... SS as manually controlled output SPI_SS_AUTO_ENVELOPE ... automatic SS “envelope” SPI_SS_AUTO_STROBE ... automatic SS strobe between words The two “AUTO” modes may be OR-combined together to achieve both “envelope” and “strobe” behavior of the SS signal.

Description: The *SPI_SET_SS_MODE* ioctl command configures the behavior of the slave-select signal when SPI operates in SPI Master mode. The SPI_SS_INPUT is typically used when mode-fault error detection is enabled (MODFEN bit is set) by the SPI_SET_MODE_FAULT ioctl command.

The SPI_SS_MANUAL_OUT may be used when software needs a full control over the SS output pin. This mode may be used as an alternative to a plain GPIO-based control over the SS pin. Use the SPI_SET_SS_OUTPUT command to control the SS output pin in software.

The SPI_SS_AUTO_ENVELOPE may be used to enable an automatic control of the SS pin. The SS output signal is generated to create the initial falling edge and final rising edge “envelope” around the data word. An idle state of the SS signal is high.

The SPI_SS_AUTO_STROBE may be used to enable an automatic control of the SS pin. The SS output signal is pulsed high between words, which adds 1.5 baud clocks to the total word period. An idle state of SS is low unless this mode is combined with the SPI_SS_AUTO_ENVELOPE parameter and then the idle state is high.

See the **56F802x/3x Peripheral Reference Manual** for more details.

This command changes the SS_DDR, SS_AUTO and SS_STRB bits of the SPI Data Size and Control Register (SPDSCR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_SET_SS_MODE* ioctl command is implemented as a macro.

Example 5-553. SPI_SET_SS_MODE

```
ioctl(SPI, SPI_SET_SS_MODE, SPI_SS_MANUAL_OUT);
```

This code configures the slave-select pin to a manual control mode.

5.16.3.30 SPI_SET_SS_OUTPUT - controls the SS output in manual mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_SS_OUTPUT,
     UWord16 param);
```

Arguments:

Table 5-627. SPI_SET_SS_OUTPUT ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	UWord16 (or generally an integer format) evaluated as zero or non-zero. You may use the following predefined constants: SPI_HIGH - set the SS output high SPI_LOW - set the SS output low

Description: The *SPI_SET_SS_OUTPUT* ioctl command sets or clears the SS output pin if SS is in configured to "manual" mode using the SPI_SET_SS_MODE ioctl command. See the **56F802x/3x Peripheral Reference Manual** for more details.

This command sets or resets the SS_DATA bit of the SPI Data Size and Control Register (SPDSCR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_SET_SS_OUTPUT* ioctl command is implemented as a macro.

Example 5-554. SPI_SET_SS_OUTPUT

```
ioctl(SPI, SPI_SET_SS_MODE, SPI_SS_MANUAL_OUT);
ioctl(SPI, SPI_SET_SS_OUTPUT, SPI_HIGH);
```

This code configures the slave-select pin for a manual control and sets the pin output high.

5.16.3.31 SPI_SET_SS_WIRED_OR_MODE - set the SS pin mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_SS_WIRED_OR_MODE,
     UWord16 param);
```

Arguments:

Table 5-628. SPI_SET_SS_WIRED_OR_MODE ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_ENABLE ... sets the SS pin to open-drain (wired-or) mode SPI_DISABLE ... sets the SS pin for high/low driven mode

Description: The *SPI*_SET_SS_WIRED_OR_MODE ioctl command sets the SS output pin mode to either open-drain mode or high/low driven mode. See the **56F802x/3x Peripheral Reference Manual** for more details.

In the open-drain mode, only the low output level is driven while the high output level is achieved using the pull-up resistor connected to the SS signal. This mode is suitable for multiple master systems.

In the high/low driven mode, the pin is actively driven high or low by the SPI module. This mode is generally used in single master systems.

This command sets or resets the SS_ODM bit of the SPI Data Size and Control Register (SPDSCR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_SET_SS_WIRED_OR_MODE* ioctl command is implemented as a macro.

Example 5-555. SPI_SET_SS_WIRED_OR_MODE

```
ioctl(SPI, SPI_SET_SS_WIRED_OR_MODE, SPI_ENABLE);
```

This code configures the slave-select pin to the open-drain mode.

5.16.3.32 SPI_OVERRIDE_SS_INPUT - override the internal SS signal with SPMSTR

Call(s):

```
void ioctl(const int *pModuleBase, SPI_OVERRIDE_SS_INPUT,
     UWord16 param);
```

Arguments:

Table 5-629. SPI_OVERRIDE_SS_INPUT ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_ENABLE ... override the internal SS signal by value of SPMSTR bit SPI_DISABLE ... revert the SS signal to be sourced from the SS pin

Description: The *SPI_OVERRIDE_SS_INPUT* ioctl command enables the SS signal to be internally overridden by a value of the SPMSTR bit. This allows the SPI module to function in slave mode, CPHA = 1, without committing the SS pin to be tied low. This bit should not be used in multi-slave systems or when CPHA = 0. In master mode a mode fault error can not be generated, therefore this command should not be used in a multi-master system. See the **56F802x/3x Peripheral Reference Manual** for more details.

This command sets or resets the SS_ORR bit of the SPI Data Size and Control Register (SPDSCR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *SPI_OVERRIDE_SS_INPUT* ioctl command is implemented as a macro.

Example 5-556. SPI_OVERRIDE_SS_INPUT

```
ioctl(SPI, SPI_OVERRIDE_SS_INPUT, SPI_DISABLE);
```

This code reverts the SS signal to a default behavior when it is sourced from the SS pin.

5.16.3.33 SPI_QUEUED_MODE - enables or disables queued operation

Call(s):

```
void ioctl(const int *pModuleBase, SPI_QUEUED_MODE,
     UWord16 param);
```

Arguments:

Table 5-630. SPI_QUEUED_MODE ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use SPI_ENABLE to enable queued operation or SPI_DISABLE to disable it.

Description: The *SPI_QUEUED_MODE* ioctl command enables or disables the receive and transmit FIFOs by modifying the FIFO_EN bit in the SPI Control 2 Register. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that for a queued operation, you still need to set-up the FIFO threshold levels by using the SPI_SET_RXFULL_CONDITION or SPI_SET_TXEMPTY_CONDITION ioctl commands.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SPI_QUEUED_MODE* ioctl command is implemented as a macro.

Example 5-557. SPI_QUEUED_MODE

```
ioctl(SPI, SPI_QUEUED_MODE, SPI_ENABLE);
```

This code enables queued mode of the SPI module.

5.16.3.34 SPI_SET_TXEMPTY_CONDITION - configures the transmit FIFO threshold in queued mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_TXEMPTY_CONDITION,
     UWord16 param);
```

Arguments:

Table 5-631. SPI_SET_TXEMPTY_CONDITION ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_TXEMPTY_WHEN_FIFO_EMPTY SPI_TXEMPTY_WHEN_1_OR_LESS_IN_FIFO SPI_TXEMPTY_WHEN_2_OR_LESS_IN_FIFO SPI_TXEMPTY_WHEN_3_OR_LESS_IN_FIFO

Description: The *SPI_SET_TXEMPTY_CONDITION* ioctl command configures the transmit FIFO threshold, which is the level at which the SPI signals to CPU that it requires new byte for transmission. The signaling is done a standard way using the Transmitter-Empty flag or interrupt. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that the queued mode must be enabled using the SPI_QUEUED_MODE ioctl command after this command, to achieve the desired “queued” functionality.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SPI_SET_TXEMPTY_CONDITION* ioctl command is implemented as a macro.

Example 5-558. SPI_SET_TXEMPTY_CONDITION

```
ioctl(SPI, SPI_SET_TXEMPTY_CONDITION,
     SPI_TXEMPTY_WHEN_FIFO_EMPTY);
```

This code sets the transmit FIFO threshold in a way that this FIFO must go completely empty before the SPI will signal TXEMPTY. In other words it means that when the TXEMPTY flag or interrupt is detected, up to four bytes (FIFO depth) can be written to the transmit register at once. See also the SPI_CAN_WRITE_DATA ioctl command.

5.16.3.35 SPI_SET_RXFULL_CONDITION - configures the receive FIFO threshold in queued mode

Call(s):

```
void ioctl(const int *pModuleBase, SPI_SET_RXFULL_CONDITION,
     UWord16 param);
```

Arguments:

Table 5-632. SPI_SET_RXFULL_CONDITION ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.
param	in	Use one of the predefined constants: SPI_RXFULL_WHEN_1_OR_MORE_IN_FIFO SPI_RXFULL_WHEN_2_OR_MORE_IN_FIFO SPI_RXFULL_WHEN_3_OR_MORE_IN_FIFO SPI_RXFULL_WHEN_4_OR_MORE_IN_FIFO

Description: The *SPI_SET_RXFULL_CONDITION* ioctl command configures the receive FIFO threshold, which is the level at which the SPI signals to CPU that the byte(s) were received and are available for reading-out of the SPI data register. The signaling is done a standard way using the Receiver-full flag or interrupt. See the **56F802x/3x Peripheral Reference Manual** for more details.

Note that the queued mode must be enabled using the SPI_QUEUED_MODE ioctl command after this command, to achieve the desired “queued” functionality.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SPI_SET_RXFULL_CONDITION* ioctl command is implemented as a macro.

Example 5-559. SPI_SET_RXFULL_CONDITION

```
ioctl(SPI, SPI_SET_RXFULL_CONDITION,
     SPI_RXFULL_WHEN_1_OR_MORE_IN_FIFO);
```

This code sets the receive FIFO threshold in a way that even a single character received causes the RXFULL flag to be set, just like in the non-queued mode. However, unlike the non-queued mode the receive FIFO accepts additional three characters (up to FIFO depth) before the receive-overrun occurs. See also the SPI_CAN_READ_DATA ioctl command.

5.16.3.36 SPI_CAN_READ_DATA - checks if data byte is available for reading

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_CAN_READ_DATA,
      NULL);
```

Arguments:

Table 5-633. SPI_CAN_READ_DATA ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_CAN_READ_DATA* ioctl command returns a non-zero value when there is at least one character received in the receive FIFO when operating in the queued mode. When a non-zero value is returned, the data may be read-out using the SPI_READ_DATA ioctl command.

Note that the queued mode must first be enabled using the SPI_QUEUED_MODE ioctl command to be able to use this command.

Returns: UWord16 for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SPI_CAN_READ_DATA* ioctl command is implemented as a macro.

Example 5-560. SPI_CAN_READ_DATA

```
UWord16 byte;

while(ioctl(SPI, SPI_CAN_READ_DATA, NULL))
{
      byte = ioctl(SPI, SPI_READ_DATA, NULL);
      ...

}
```

This code reads all bytes from the SPI receive FIFO.

5.16.3.37 SPI_CAN_WRITE_DATA - checks if a free slot is available in transmit FIFO

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_CAN_WRITE_DATA,
      NULL);
```

Arguments:

Table 5-634. SPI_CAN_WRITE_DATA ioctl call arguments

*pModuleBase	in	The SPI module identifier, use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_CAN_WRITE_DATA* ioctl command returns a non-zero value when there is at least one free position left in the transmit FIFO when operating in the queued mode. When a non-zero value is returned, at least one data byte may be written to the SPI using the SPI_WRITE_DATA ioctl command.

Note that the queued mode must first be enabled using the SPI_QUEUED_MODE ioctl command to be able to use this command.

Returns: UWord16 for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *SPI_CAN_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-561. SPI_CAN_WRITE_DATA

```
if(ioctl(SPI, SPI_CAN_WRITE_DATA, NULL))
{
      ioctl(SPI, SPI_WRITE_DATA, *pNextByte++);

}
```

This code writes one data byte into the transmit FIFO if possible.

5.16.3.38 read - block read from SPI module

Call(s):

```
void read (const int *pModuleBase, NON_BLOCKING,
                                   int *address, int size);
void read (const int *pModuleBase, BLOCKING,
                                   int *address, int size);
```

Arguments:

Table 5-635. read function call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *read* function is implemented as a macro and calls related *read* functions. There are two modes of operation, which are controlled by the *mode* parameter:

non-blocking mode: The *read* function in non-blocking mode initializes some internal status variables, pointers and associated flags and read operations can start. The *read* function immediately returns to the caller and the end of read operations can be tested with the SPI_GET_STATUS command. All operations are carried out in the interrupt service routine, which must be installed properly, see Table 5-636. The *read* function enables SPI receiver and error interrupts and this interrupt is still enabled after finishing of the read operations. If the *read* function in blocking mode is called after, these interrupts should be disabled by SPI_RX_FULL_INT and SPI_ERROR_INT commands.

blocking mode: The *read* function in blocking mode uses a polling technique, interrupts are not used. The *read* function in blocking mode waits until all characters are read from the SPI module. In master SPI mode the parameter size (number of characters to receive) can be only 1 when blocking mode is used for *write* function. It is because *write* function in blocking mode waits till all characters are transmitted and successive read function call cannot read all received characters because hardware receive buffer is only 1 word long (transmit and receive operation works concurrently on SPI). It is possible to use parameter size > 1 when in slave SPI mode (to receive only) or when *write* function is called in non-blocking mode.

Returns: None.

Range Issues: Parameter *size* must greater than 1.

Special Issues: non-blocking mode: If the non-blocking mode of the *read* function is used, interrupt functions of the SPI driver must be installed to obtain proper functionality of the *read* function (See Table 5-636).

Table 5-636. Interrupt routines of the SPI driver for the read function in non-blocking mode

void Spi*x*ReceiveISR(void)	This function is dedicated to service SPI *x* Receiver/Error Interrupt and must be installed at proper interrupt vector (See Example 5-566).

Note: Symbol *"x"* in the Table 5-636 should be replaced with the number of the SPI module e.g. *Spi0ReceiveISR*.

non-blocking mode: None.

Design/Implementation: The non-blocking and blocking mode of the *read* macro are implemented as a function call.

5.16.3.39 write - block write to SPI module

Call(s):

```
void write (const int *pModuleBase, NON_BLOCKING,
                                    int *address, int size);
void write (const int *pModuleBase, BLOCKING,
                                    int *address, int size);
```

Arguments:

Table 5-637. write function call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *write* function is implemented as a macro and calls related *write* functions for non-blocking and blocking mode, depending on the *mode* parameter.

non-blocking mode: The *write* function in non-blocking mode initializes some internal status variables, pointers and associated flags and write operation starts (if in MASTER mode). A *write* function immediately returns to the caller and the end of write operations can be tested with the SPI_GET_STATUS command. All operations are carried out in the interrupt service routine, which must be installed properly, see Table 5-638.

blocking mode: The *write* function in blocking mode uses a polling technique, interrupts are not used. The *write* function in blocking mode waits until all characters are written to the SPI.

Returns: None.

Range Issues: Parameter *size* must greater than 1.

Special Issues:

non-blocking mode: If the non-blocking mode of the *write* function is used, interrupt functions of the SPI driver must be installed to obtain proper functionality of the *write* function (See Table 5-638).

Table 5-638. Interrupt routines of the SPI driver for the write function in non-blocking mode

void Spi*x*TxEmptyISR(void)	This function is dedicated to service SPI *x* Transmitter Empty Interrupt and must be installed at proper interrupt vector (See Example 5-566).

Note: Symbol *"x"* in the Table 5-638 should be replaced with the number of the SPI module e.g. *Spi0TxEmptyISR*.

non-blocking mode: None.

Design/Implementation: The non-blocking and blocking mode of the *write* macro are implemented as a function call.

5.16.3.40 SPI_CLEAR_EXCEPTION - clear read/write function exception if exists

Call(s):

```
void ioctl(const int *pModuleBase, SPI_CLEAR_EXCEPTION, NULL);
```

Arguments:

Table 5-639. SPI_CLEAR_EXCEPTION ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_CLEAR_EXCEPTION* ioctl command clears software error flags, generated by *read/write* functions. The *SPI_CLEAR_EXCEPTION* ioctl command modifies the internal status variable of the SPI driver.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_CLEAR_EXCEPTION* ioctl command is implemented as a macro.

Example 5-562. SPI_CLEAR_EXCEPTION

```
ioctl(SPI_0, SPI_CLEAR_EXCEPTION, NULL);
```

This code clears an exception generated by read/write functions for the SPI 0 module.

5.16.3.41 SPI_GET_STATUS - get read/write function operation status

Call(s):

```
UWord16 ioctl(const int *pModuleBase, SPI_GET_STATUS, NULL);
```

Arguments:

Table 5-640. SPI_GET_STATUS ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_GET_STATUS* ioctl command reads the internal status variable of the SPI driver. See e.g. Example 5-563 for more details.

Returns: The *SPI_GET_STATUS* ioctl command returns the SPI_STATUS_EXCEPTION_EXIST flag, the SPI_STATUS_WRITE_INPROGRESS flag, the SPI_STATUS_READ_INPROGRESS flag, the SPI_EXCEPTION_BUFFER_OVERFLOW flag, the SPI_EXCEPTION_MODE_FAULT flag, the SPI_EXCEPTION_OVERRUN flag.

The SPI_STATUS_EXCEPTION_EXIST flag is set, when a software or a hardware exception occurred during an operation of the SPI module and a read or write function was performed.

If the SPI_STATUS_EXCEPTION_EXIST and the SPI_EXCEPTION_BUFFER_OVERFLOW flags are set, it means that some extra characters were received by the SPI module and they are lost.

If the SPI_STATUS_EXCEPTION_EXIST and the SPI_EXCEPTION_MODE_FAULT flags are set, it means that a “mode fault” hardware error occurred during SPI module operation. This error condition can be mainly caused by an improper setting of the SPI module. For more info see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

If the SPI_STATUS_EXCEPTION_EXIST and the SPI_EXCEPTION_OVERRUN flags are set, it means that an “overrun” hardware error occurred during SPI module operation. This error condition happens when received data are not fetched from the SPI RX register before a new character is received. This can be caused by an improper installation of the RX interrupt service routine for non-blocking mode (or by disabled interrupts). For blocking mode it can happen when the number of received characters is higher than the number defined for read function. For more info, see the **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**.

If the SPI_STATUS_WRITE_INPROGRESS or the SPI_STATUS_READ_INPROGRESS flags are set, the *read* or the *write* function has not finished its operation. After finishing a read or write operation, the respective flag is cleared. See also Chapter 5.16.3.38 and Chapter 5.16.3.39.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_GET_STATUS* ioctl command is implemented as a macro.

Example 5-563. SPI_GET_STATUS

```
tmp = ioctl(SPI_0, SPI_GET_STATUS, NULL);
```

This code reads the *read*/*write* function operation status and stores it to variable *tmp*.

5.16.3.42 SPI_WRITE_CANCEL - cancel non-blocking write operation

Call(s):

```
void ioctl(const int *pModuleBase, SCI_WRITE_CANCEL, NULL);
```

Arguments:

Table 5-641. SPI_WRITE_CANCEL ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_WRITE_CANCEL* ioctl command cancels a non-blocking write operation if it is not completed.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_WRITE_CANCEL* ioctl command is implemented as a macro.

Example 5-564. SPI_WRITE_CANCEL

```
write(SPI_0, NON_BLOCKING, buffer, 10);
ioctl(SPI_0, SPI_WRITE_CANCEL, NULL);
```

This code cancels a non-blocking write operation on the SPI 0 module. It means that a write function can be called again before finishing previous write operations.

5.16.3.43 SPI_READ_CANCEL - cancel non-blocking read operation

Call(s):

```
void ioctl(const int *pModuleBase, SPI_READ_CANCEL, NULL);
```

Arguments:

Table 5-642. SPI_READ_CANCEL ioctl call arguments

*pModuleBase	in	The SPI module identifier. Use SPI_0 or SPI_1. Note that SPI is equivalent to SPI_0 on devices with only one SPI module.

Description: The *SPI_READ_CANCEL* ioctl command cancels a non-blocking read operation if it is not completed. See e.g. Example 5-565 for more details.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *SPI_READ_CANCEL* ioctl command is implemented as a macro.

Example 5-565. SPI_READ_CANCEL

```
read(SPI_0, NON_BLOCKING, buffer, 10);
/* ... */
ioctl(SPI_0, SPI_READ_CANCEL, NULL);
read(SPI_0, NON_BLOCKING, buffer, 5);
```

This code cancels non-blocking read operation on the SPI 0 module during receive operation. It means that a read function can be called again before finishing previous read operations. After that it runs another read operation for 5 words.

5.16.4 SPI Driver Application

The SPI driver application is designed for Freescale/Motorola EVMs and intended to illustrate the usage of this driver by a real example and also to verify the functionality by transmission to the SPI peripheral or to another MCU/DSP.

The SPI driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM* directory and consists of the application project *spi_demo.mcp* and the source code for the application *main.c*.

Example 5-566. SPI Driver Sample Application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Tue, 13/Feb/2007, 13:17:39
*
**************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02030010L

/*.
    OCCS Configuration
-----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
```

```
                          TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                          CAN: Enable , ADC A: Enable , ADC B: Enable
                          EMI: Enable
.*/
#define SIM_GPS_INIT                    0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                0x0005
#define SEMI_CSBAR1_INIT                0x0005
#define SEMI_CSBAR2_INIT                0x0005
#define SEMI_CSOR0_INIT                 0x1FC3
#define SEMI_CSOR1_INIT                 0x1BA3
#define SEMI_CSOR2_INIT                 0x1DA3

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: TXD1 , PullUp: Enable ,
    Pin  7: Function: RXD1 , PullUp: Enable ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT                 0x1FC1

/*.
    GPIO_E Configuration
-----------------------------------------
    Pin  0: Function: TXD0 , PullUp: Enable ,
    Pin  1: Function: RXD0 , PullUp: Enable ,
    Pin  2: Function: A6 , PullUp: Enable ,
```

```
    Pin  3: Function: A7 , PullUp: Enable ,
    Pin  4: Function: SCLK0 , PullUp: Enable ,
    Pin  5: Function: MOSI0 , PullUp: Enable ,
    Pin  6: Function: MISO0 , PullUp: Enable ,
    Pin  7: Function: GPIO , Direction: Output , Init.Value: High - 1 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  8: Function: TC0 , PullUp: Enable ,
    Pin 10: Function: TD0 , PullUp: Enable ,
    Pin 11: Function: TD1 , PullUp: Enable ,
.*/
#define GPIO_E_DDR_INIT                  0x0080
#define GPIO_E_PER_INIT                  0x3F7F
#define GPIO_E_DR_INIT                   0x0080

/*.
    SPI_0 Configuration
-----------------------------------------
    SPI Module Enable: Enable
    Clock frequency: 3.75 MHz
    SPI Mode: Master
    SCLK Polarity: Falling edge
    SCLK Phase: Transmission started by edge on SS pin
    Shift order: MSB first
    Wired OR mode: Normal
    SS Mode Fault Enable: Enable
    Interrupts: Receiver: Disable
                Error: Disable
                Transmitter: Disable
.*/
#define SPI_0_SPSCR_INIT                 0x6520


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-567. SPI Driver Sample Application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION:  SPI driver demo program - communicates with MAXIM MAX5251
*               quad 10-bit DAC. This simple application just sets the four
*               DAC outputs of MAX5251 connected to SPI_0 :
*
*               DAC channel0 = 25%      (0.675V with 2.7V Ref)
*               DAC channel1 = 60%      (1.62V with 2.7V Ref)
*               DAC channel2 = 40%      (1.08V with 2.7V Ref)
*               DAC channel3 = 75%      (2.025V with 2.7V Ref)
*
* NOTE: MAXIM MAX5251 D/A Converter is optional on the evaluation board
*
* TARGET: MC56F83xx devices
*
*****************************************************************************/

#include "qs.h"

#include "cop.h"
#include "sys.h"
#include "gpio.h"
#include "spi.h"

/* commands for MAX5251 */
#define DAC_LOAD_ALL_WITH_UPDATE    0x8000
#define DAC_LOAD_A                  0x1000
#define DAC_LOAD_B                  0x5000
#define DAC_LOAD_C                  0x9000
#define DAC_LOAD_D                  0xD000
#define DAC_UPDATE_ALL              0x4000
#define DAC_LOAD_A_UPDATE           0x3000
#define DAC_LOAD_B_UPDATE           0x7000
#define DAC_LOAD_C_UPDATE           0xB000
#define DAC_LOAD_D_UPDATE           0xF000

/* board-specific LEDs and device-specific timer module */
#include "../board.h"

/* forward prototypes */
void device_init(void);
UWord16 WriteDAC5251(UWord16 cmd, Frac16 w);

/*
 *  The main
 */

void main (void)
{
    /* initialize SYS, COP and pins */
    device_init();

    /* SPI initialization according appconfig.h */
    ioctl(SPI_0, SPI_INIT, NULL);
    ioctl(GPIO_SPI, GPIO_SET_PIN, SPI_SS);
```

```
    /* write all 4 DAC registers and update DAC outputs */
    WriteDAC5251(DAC_LOAD_ALL_WITH_UPDATE, 0);

    /* load to channel0 input reg. (0.675V with 2.7V Ref) */
    WriteDAC5251(DAC_LOAD_A, FRAC16(0.25));

    /* load to channel1 input reg. (1.62V with 2.7V Ref) */
    WriteDAC5251(DAC_LOAD_B, FRAC16(0.6));

    /* Example of dynamic SPI reconfiguration */
    /* Just be aware that for most other SPI settings it is neccesary
       to disable SPI before reconfiguration by SPI_DISABLE */
    ioctl(SPI_0, SPI_SET_BAUD_DIV, SPI_DIV8);

    /* load to channel2 input reg. (1.08V with 2.7V Ref) */
    WriteDAC5251(DAC_LOAD_C, FRAC16(0.4));

    /* load to channel3 input reg. (2.025V with 2.7V Ref) */
    WriteDAC5251(DAC_LOAD_D, FRAC16(0.75));

    /* update all DAC output registers from respective input registers */
    WriteDAC5251(DAC_UPDATE_ALL, 0);

    /* endless loop */
    while(1)
    {
        /* keep the watchdog counter clear (in case it is enabled) */
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
    }
}


/*
 *  WriteDAC5251 function
 *
 *    WriteDAC5251 - writes value to DAC registers
 *
 *    Returns: word received from SPI bus
 *
 *    cmd - command for DAC - type of operation (see MAX5251 datasheet)
 *    w   - DAC register value, format Frac16
 *          output_voltage = w*Uref
 */

UWord16 WriteDAC5251(UWord16 cmd, Frac16 w)
{
    /* /CS DAC "L" */
    ioctl(GPIO_SPI, GPIO_CLEAR_PIN, SPI_SS);

    /* write to SPI bus, top 4 bits create DAC command, */
    /* the middle 10 bits is DAC value, lowest 2 bits are zero */
    ioctl(SPI_0, SPI_WRITE_DATA, cmd | (((UWord16)w>>3) & 0x0FFC));

    /* waits to SPI write end - it tests RX full flag */
    while (!ioctl(SPI_0, SPI_GET_RX_FULL, NULL))
    {
        /* possible to test the other events */
    }

    /* inserted delay for timing requirements of MAX5251 */
    //archDelay(3);

    /* /CS DAC "H" */
    ioctl(GPIO_SPI, GPIO_SET_PIN, SPI_SS);
```

```
    /* receive return value from SPI bus */
    return ioctl(SPI_0, SPI_READ_DATA, NULL);
}

/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.17 IIC Driver (MC56F801x,MC56F800x)

This section describes the DSP56800E_Quick_Start API for the Inter-Integrated Circuit (IIC) Bus Controller which is implemented on MC56F801x and MC56F800x family of microcontrollers. The functionality of the IIC module itself is described in the **56F8000 Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

The following sections describe the IIC driver software which provides the low level API to the IIC hardware.

5.17.1 Introduction

The Inter-IC Bus (IIC or I2C) is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange between devices. Being a two-wire device, the IIC Bus minimizes the need for large numbers of connections between devices, and eliminates the need for an address decoder. This bus is suitable for applications requiring occasional communications over a short distance between a number of devices. It also provides flexibility, allowing additional devices to be connected to the bus for further expansion and system development. The interface will operate at baud rates of up to 100kbps with maximum capacitive bus loading. With reduced bus slew rate, the device is capable of operating at higher baud rates, up to a maximum of system_clock/20. The module can operate up to a baud rate of 400kbps provided the IIC bus slew rate is less than 100ns.

The IIC module of the 56F800E hybrid controllers has the following key features:

- Compatible with I2C Bus standard
- Multi-master operation
- Software programmable for one of 256 different serial clock frequencies
- Software selectable acknowledge bit
- Interrupt driven byte-by-byte data transfer
- Arbitration lost interrupt with automatic mode switching from master to slave
- Calling address identification interrupt
- Start and stop signal generation/detection
- Repeated start signal generation/detection
- Acknowledge bit generation/detection
- Bus busy detection

5.17.1.1 IIC Bus Transactions

On the IIC Bus, each node may operate as a master, slave or may be dynamically switched between master and slave modes. At any moment in time, only one master is allowed to control the transactions on the bus while other nodes are required to be either de-activated or to operate in the slave mode. The roles can be changed after one transaction is complete.

From the IIC bus node's standpoint, the IIC module is either in Master or Slave mode in association with being transmitting or receiving data. An IIC transaction begins when the Master transmits the address byte and the addressed (Slave) node acknowledges it. In the first byte, the Master also specifies whether the transaction is Read-from-slave or Write-to-slave.

For the Write transactions, the Master remains in transmitting mode and continues with the data bytes transmission being continuously acknowledged by the Slave. The master finishes the transaction either by releasing the bus by reverting to the Slave mode (generates the STOP condition on the bus), or by re-starting a new transaction to a different slave (repeated START condition). Slave may abort the Master Write transaction by not acknowledging a data byte received - master is then required either to STOP or re-START the transaction.

For the Read transactions, the Master switches to a Receive mode immediately after the address byte is transmitted and acknowledged by the Slave. Then the Master waits and acknowledges a Slave transmission. The Master finishes the transaction by not-acknowledging the last byte received from the Slave. Slave then releases the bus for the Master, which in-turn generates the STOP or repeated START condition.

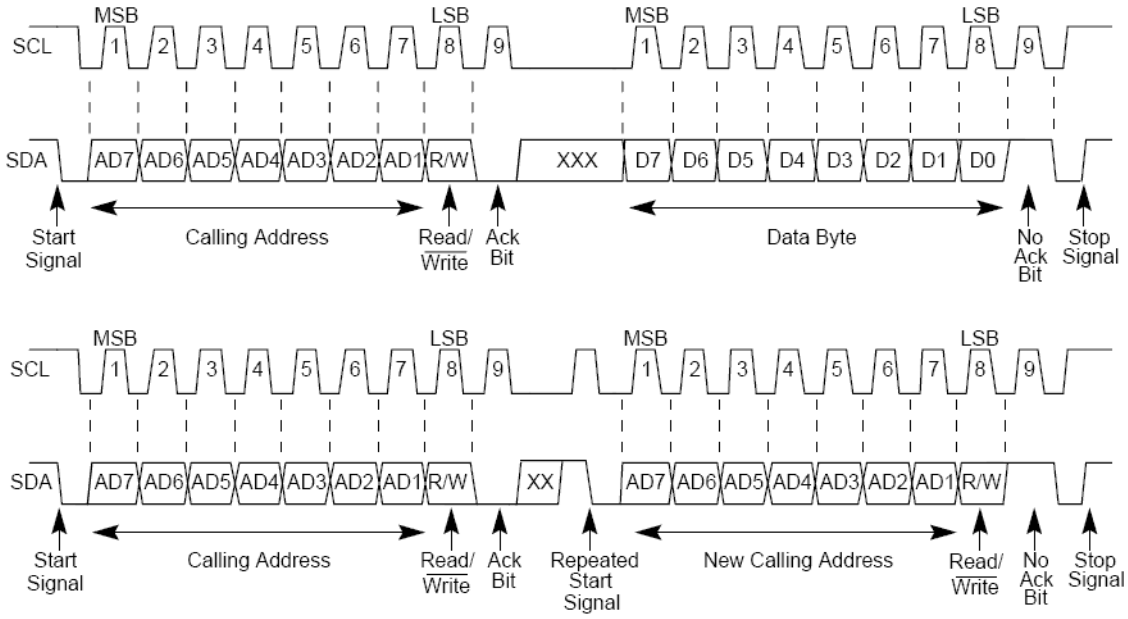


Figure 5-2. IIC Bus Transmission Signals

5.17.2 Quick Reference

This section defines the terms and formulas used later in Section 5.17.3.

Table 5-643. IIC Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
IIC (IIC_BASE)	0xF120	0xF0D0	N/A (different module)	N/A

5.17.2.1 API Definition

The following header files are needed in order to use the IIC device driver:

Required Header File(s):

```
#include "qs.h"
#include "iic.h"
```

The following information may be found in the header file *iic.h*.

Public Data Structure(s):

none

General-Use Macros and inline functions:

none

5.17.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static IIC module configuration. These symbols can be put in the project-specific configuration file *appconfig.h* and are applied upon issuing the *IIC_INIT* command.

Table 5-644. IIC Configuration Items for appconfig.h MC56F801x

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
IIC_CTRL_INIT	UWord16	Initial value for selected control bits of the IIC Control Register (CTRL). The bits are written to the CTRL as last, after all other IIC registers are initialized.	Reset value used, the IIC module remains disabled in reset-state.
IIC_FDIV_INIT	UWord16	Initial value of the IIC Frequency Divider Register (FDIV).	Register not initialized. Reset value used.
IIC_NFILT_INIT	UWord16	Initial value of the IIC Noise Filter Register (NFILT).	Register not initialized. Reset value used.
IIC_ADDR_INIT	UWord16	Initial value of the IIC Address Register (ADDR).	Register not initialized. Reset value used.

Table 5-645. IIC Configuration Items for appconfig.h MC56F800x

SYMBOL	TYPE	DESCRIPTION	WHEN UNDEFINED
IIC_IICC1_INIT	UWord16	Initial value for selected control bits of the IIC Control Register 1 (IICC1). The bits are written to the IICC1 as last, after all other IIC registers are initialized.	Reset value used, the IIC module remains disabled in reset-state.
IIC_IICC2_INIT	UWord16	Initial value for selected control bits of the IIC Control Register 2 (IICC2).	Register not initialized. Reset value used.
IIC_IICF_INIT	UWord16	Initial value of the IIC Frequency Divider Register (IICF).	Register not initialized. Reset value used.
IIC_IICA1_INIT	UWord16	Initial value of the IIC Address Register 1 (IICA1).	Register not initialized. Reset value used.
IIC_IICA2_INIT	UWord16	Initial value of the IIC Address Register 2 (IICA2).	Register not initialized. Reset value used.
IIC_IICSLTL_INIT	UWord16	Initial value for the Low Time Out Register Low (IICSLTL).	Register not initialized. Reset value used.
IIC_IICSLTH_INIT	UWord16	Initial value for the Low Time Out Register High (IICSLTH).	Register not initialized. Reset value used.
IIC_IICSMB_INIT	UWord16	Initial value for the Control and Status Register (IICSMB).	Register not initialized. Reset value used.

5.17.2.3 API Specification

This section briefly describes the API macros and functions.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call (macro) is generally represented in the following forms:

```
Word32 ioctl(const int *pModuleBase, void Cmd, void param);
Word16 ioctl(const int *pModuleBase, void Cmd, void param);
void ioctl(const int *pModuleBase, void Cmd, void param);
```

Description: The *ioctl* call "changes" IIC device modes or accesses the IIC register(s). Keep in mind that *ioctl* is treated as macro that is in result mostly compiled to an optimal inline code.

Arguments:

Table 5-646. IIC Driver Arguments - ioctl

pModuleBase	in	IIC module identifier. Use IIC.
cmd	in	Command names found in *iic.h*. See Table 5-647.
pParam	in, inout	Used to pass the relevant data to ioctl

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-647. IIC Module ioctl Commands

Cmd	pParam	Return	Description
IIC_INIT	NULL	None	Initializes the IIC module using the *appconfig.h* values.
IIC_SET_ADDRESS	UWord16	None	Set the slave address of the IIC Bus node

Table 5-647. IIC Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
IIC_GET_ADDRESS	NULL	UWord16	Get the currently valid slave address of the IIC Bus node.
IIC_SET_PRESCALER	UWord16	None	Write the IIC Frequency Divider Register. See the IIC module description for the register format.
IIC_I_BUS	IIC_ENABLE / IIC_DISABLE	None	Enable/disable the IIC module.
IIC_I_BUS_INT	IIC_ENABLE / IIC_DIABLE	None	Enable/disable the IIC module interrupt to be generated to the core.
IIC_MASTER_SLAVE_MODE	IIC_MASTER / IIC_SLAVE	None	Switch the IIC module into Master or Slave mode.
IIC_GET_MASTER_MODE	NULL	UWord16	Return non-zero if IIC Master mode is set.
IIC_TX_RX_MODE	IIC_TRANSMIT / IIC_RECEIVE	None	Select the direction of immediate data transfers.
IIC_GET_TX_MODE	NULL	UWord16	Return non-zero if transmit mode is set.
IIC_TX_ACK	IIC_ACK / IIC_NO_ACK	None	Enable/disable the automatic transmitting of the Acknowledge signal for both master/slave receivers.
IIC_REPEAT_START	NULL	None	Generate a repeated START condition on the IIC Bus.
IIC_WRITE_CONTROL_REG	UWord16	None	Write to the IIC Bus Control Register.
IIC_READ_CONTROL_REG	NULL	UWord16	Read the IIC-Bus Control Register as UWord16.
IIC_WRITE_DATA	UWord16	None	Write IIC Data I/O Register.
IIC_READ_DATA	NULL	UWord16	Read IIC Data I/O Register.
MC56F801x only: IIC_SET_NOISE_FILTER	UWord16	None	Write IIC Noise Filter bits.
IIC_READ_STATUS_REG	NULL	UWord16	Read IIC Status Register
IIC_TEST_STATUS_REG	IIC_TRANSFER_COMPLETE | IIC_ADDRESSED_AS_SLAVE | IIC_BUS_BUSY | IIC_ARBITRATION_LOST | IIC_SLAVE_TRANSMIT | IIC_IBUS_INT | IIC_RX_ACK	UWord16	Test IIC Status Register for the selected bits. Returns non-zero if any of bits selected in the parameter are set.

Table 5-647. IIC Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
IIC_GET_TRANSFER_COMPLETE *IIC_GET_ADDRESSED_AS_SLAVE* *IIC_GET_BUS_BUSY* *IIC_GET_ARBITRATION_LOST* *IIC_GET_SLAVE_TRANSMIT* *IIC_GET_I_BUS_INT* *IIC_GET_RX_ACK*	*NULL*	*UWord16*	*Single-bit testing in the IIC Status Register. Returns non-zero if the particular bit is set.* *These commands can be used instead of the command above when testing a single status bit.*
IIC_CLEAR_ARBITRATION_LOST	NULL	None	Clear the Arbitration Lost status.
IIC_CLEAR_I_BUS_INT	NULL	None	Clear the IIC Bus Interrupt flag.
MC56F800x only: IIC_SET_GENERAL_CALL_ADDRESS	IIC_ENABLE / IIC_DISABLE	None	Enables / disables general call address.
MC56F800x only: IIC_SET_ADDRESS_EXTENSION_MODE	IIC_7BIT / IIC_10BIT	None	Select 7-bit / 10-bit address scheme.
MC56F800x only: IIC_SET_ADDRESS_EXTENSION	UWord16 (0..7)	None	Sets high part of the IIC address in 10-bit address scheme.
MC56F800x only: IIC_SET_10BIT_ADDRESS	UWord16 (0..1023)	None	Sets the IIC 10-bit slave address.
MC56F800x only: IIC_GET_10BIT_ADDRESS	NULL	UWord16	Reads the IIC 10-bit slave address.
MC56F800x only: IIC_SET_FAST_ACK_NACK	IIC_ENABLE / IIC_DISABLE	None	Enables / disables fast set up ACK/NACK response.
MC56F800x only: IIC_SET_SMBUS_RESPONSE_ADDRESS	IIC_ENABLE / IIC_DISABLE	None	Enables / disables SMBUS alert response address.
MC56F800x only: IIC_SET_SECOND_IIC_ADDRESS	IIC_ENABLE / IIC_DISABLE	None	Enables / disables SMBus device default address.
MC56F800x only: IIC_SET_TIME_OUT_CLOCK	IIC_DIV1 / IIC_DIV64	None	Selects the clock source of Time Out Counter.
MC56F800x only: IIC_TEST_TIMEOUT_FLAGS	IIC_LOW_TIMEOUT | IIC_HIGH_TIMEOUT	UWord16	Tests selected timeout flags.
MC56F800x only: IIC_CLEAR_LOW_TIMEOUT_FLAG	NULL	None	Clears low timeout flag.
MC56F800x only: IIC_SET_SMBUS_ADDRESS	UWord16 (0..127)	None	Sets SMBus address.
MC56F800x only: IIC_GET_SMBUS_ADDRESS	NULL	UWord16	Reads and returns SMBus address.
MC56F800x only: IIC_WRITE_SCL_LOW_TIMEOUT	UWord16(0..65535)	None	Writes value into the SCL low time-out registers.

Table 5-647. IIC Module ioctl Commands (Continued)

Cmd	pParam	Return	Description
MC56F800x only: IIC_READ_SCL_LOW_TIMEOUT	NULL	UWord16	Reads and returns value from the SCL low timeout registers.

5.17.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.17.3.1 IIC_INIT - initialize IIC module

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INIT, NULL);
```

Arguments:

Table 5-648. IIC_INIT ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: This command initializes the IIC module according to the values from *appconfig.h* configuration file.

The last of the IIC registers initialized by IIC_INIT is the IIC Control Register (CTRL on MC56F801x, IICC1 on MC56F800x). When the IBEN (ICEN) bit is set in the initialization value, the IIC module is enabled and made ready to be used in the application.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_INIT* ioctl command is implemented as a function call.

Example 5-568. IIC_INIT

```
ioctl(IIC, IIC_INIT, NULL);
```

This code calls the driver initialization function.

5.17.3.2 IIC_SET_ADDRESS - set the IIC slave address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_ADDRESS,
           UWord16 param);
```

Arguments:

Table 5-649. IIC_SET_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	The 7-bit address (0...127) to be used in the slave mode.

Description: The *IIC_SET_ADDRESS* ioctl command can be used when changing the IIC node address. The IIC address is used in the slave mode only to detect it is being addressed in an IIC transaction. Typically, it is not necessary to change the slave address once it is set e.g. by the IIC_INIT call.

Note that it is forbidden for the IIC module to address itself by the Master transaction. If the node needs to operate in a Master mode, it is required the address is set to a value not used on the bus.

This command writes directly into the ADR bit-field in the ADDR register on MC56F801x or IICA1 register on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_SET_ADDRESS* ioctl command is implemented as a macro.

Example 5-569. IIC_SET_ADDRESS

```
ioctl(IIC, IIC_SET_ADDRESS, 10);
```

This code sets the IIC slave address to 10.

5.17.3.3 IIC_GET_ADDRESS - get the currently active slave address

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_GET_ADDRESS, NULL);
```

Arguments:

Table 5-650. IIC_GET_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: Use this command to read the IIC address which is currently active for the slave operation.

Returns: The IIC address of the node in range 0..127 as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_GET_ADDRESS* ioctl command is implemented as a macro.

Example 5-570. IIC_GET_ADDRESS

```
UWord16 a;

a = ioctl(IIC, IIC_GET_ADDRESS, NULL);
```

This code reads the actual slave address.

5.17.3.4 IIC_SET_PRESCALER - write the IIC Frequency Divider Register

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-651. IIC_SET_PRESCALER ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	The byte value to be written to IIC Frequency Divider Register

Description: The *IIC_SET_PRESCALER* ioctl command writes directly into the FDIV register, which controls several prescalers and delays applied to the IIC bit-sampling and bit-transmission process. All bit-fields of the register value are treated as the indexes to the divider/delay value tables so it is not always a trivial task to derive a value needed in the application. For that reason, it is more convenient to use the Graphical Configuration Tool to calculate the suitable register value.

Typically, the frequency divisor value is not necessary to be changed once it is set e.g. by the IIC_INIT call.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_SET_PRESCALER* ioctl command is implemented as a macro.

Example 5-571. IIC_SET_PRESCALER

```
ioctl(IIC, IIC_SET_PRESCALER, 0xBF);
```

This code sets the following IIC Bus parameters:

- SCL Divider: 15360 clocks
- SDA Hold: 2052 clocks
- SCL Hold (Start): 7672 clocks
- SCL Hold (Stop): 7684 clocks

5.17.3.5 IIC_I_BUS - enable/disable the IIC module

Call(s):

```
void ioctl(const int *pModuleBase, IIC_I_BUS,
           UWord16 param);
```

Arguments:

Table 5-652. IIC_I_BUS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter to enable or disable the IIC module. Use one of the following constants: IIC_ENABLE - IIC module enabled IIC_DISABLE - IIC module disabled

Description: The *IIC_I_BUS* ioctl command enables or disables the IIC module as a whole.

This command writes directly into the IBEN bit of the CTRL register on MC56F801x or IICC1 on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_I_BUS* ioctl command is implemented as a macro.

Example 5-572. IIC_I_BUS

```
ioctl(IIC, IIC_I_BUS, IIC_ENABLE);
```

This code enables the IIC module.

5.17.3.6 IIC_I_BUS_INT - enable/disable the IIC bus interrupt

Call(s):

```
void ioctl(const int *pModuleBase, IIC_I_BUS_INT,
           UWord16 param);
```

Arguments:

Table 5-653. IIC_I_BUS_INT ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter to enable or disable the IIC bus interrupt processing. Use one of the following constants: IIC_ENABLE - interrupt enabled IIC_DISABLE - interrupt disabled

Description: The *IIC_I_BUS_INT* ioctl command enables or disables the IIC interrupt to propagate to the processor core. The core receives the interrupt provided if it is enabled in the INTC Interrupt Controller module and also when interrupts are enabled globally in the core Status Register (SR).

This command writes directly into the IBIE bit of the CTRL register on MC56F801x or IICC1 on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_I_BUS_INT* ioctl command is implemented as a macro.

Example 5-573. IIC_I_BUS_INT

```
ioctl(IIC, IIC_I_BUS_INT, IIC_ENABLE);
```

This code enables the interrupt of the IIC module.

5.17.3.7 IIC_MASTER_SLAVE_MODE - select IIC master or slave mode

Call(s):

```
void ioctl(const int *pModuleBase, IIC_MASTER_SLAVE_MODE,
           UWord16 param);
```

Arguments:

Table 5-654. IIC_MASTER_SLAVE_MODE ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter specifying the IIC mode to be set. Use one of the pre-defined constants: IIC_MASTER - set IIC master mode IIC_SLAVE - set IIC slave mode

Description: The *IIC_MASTER_SLAVE_MODE* ioctl command selects the mode the IIC module operates in. When used as a master mode in a single-master application, this command may be used once after the IIC initialization to set the Master mode. In a multi-master application, the IIC is typically set to Slave (listen) mode by default and is momentarily switched to the Master mode when needed.

The Master/Slave mode selection should not be confused with the Receive/Transmit selection. See Section 5.17.1.1 on page -773 for more details.

This command writes directly into the MS/$\overline{SL}$ bit of the CTRL register on MC56F801x or IICC1 on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_MASTER_SLAVE_MODE* ioctl command is implemented as a macro.

Example 5-574. IIC_MASTER_SLAVE_MODE

```
ioctl(IIC, IIC_MASTER_SLAVE_MODE, IIC_MASTER);
```

This code switches the IIC module into the master mode.

5.17.3.8 IIC_GET_MASTER_MODE - test master mode is active

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_GET_MASTER_MODE, NULL);
```

Arguments:

Table 5-655. IIC_GET_MASTER_MODE ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_GET_MASTER_MODE* ioctl command tests the MS/$\overline{SL}$ bit in the IIC Control Register and returns non-zero if the Master mode is set.

Returns: Zero/Nonzero as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x and MC56F800x.

Design/Implementation: The *IIC_GET_MASTER_MODE* ioctl command is implemented as a macro.

Example 5-575. IIC_GET_MASTER_MODE

```
if(ioctl(IIC, IIC_GET_MASTER_MODE, NULL))
{
      // ... master mode processing
}
```

This code tests the IIC is in master mode.

5.17.3.9 IIC_TX_RX_MODE - select a direction of data transfers

Call(s):

```
void ioctl(const int *pModuleBase, IIC_TX_RX_MODE,
           UWord16 param);
```

Arguments:

Table 5-656. IIC_TX_RX_MODE ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter specifying the IIC data transfer mode to be set. Use one of the predefined constants: IIC_TRANSMIT - set IIC transmission mode IIC_RECEIVE - set IIC reception mode

Description: Using the *IIC_TX_RX_MODE* ioctl command the application controls the direction of IIC communication. The Receive/Transmit mode selection should not be confused with the Master/Slave selection. See Section 5.17.1.1 on page -773 for more details.

This command writes directly the TX/$\overline{RX}$ bit of the CTRL register on MC56F801x or IICC1 on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_TX_RX_MODE* ioctl command is implemented as a macro.

Example 5-576. IIC_TX_RX_MODE

```
ioctl(IIC, IIC_TX_RX_MODE, IIC_TRANSMIT)
```

This code switches the mode into the transmitting mode.

5.17.3.10 IIC_GET_TX_MODE - test transmission mode is active

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_GET_TX_MODE, NULL);
```

Arguments:

Table 5-657. IIC_GET_TX_MODE ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_GET_TX_MODE* ioctl command tests the TX/$\overline{RX}$ bit in the IIC Control Register and returns non-zero if the Transmit mode is set.

Returns: Zero/Nonzero as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_GET_TX_MODE* ioctl command is implemented as a macro.

Example 5-577. IIC_GET_TX_MODE

```
if(ioctl(IIC, IIC_GET_TX_MODE, NULL))
{
      // ... transmit mode processing
}
```

This code tests whether the transmission mode is active.

5.17.3.11 IIC_TX_ACK - enable/disable generating of the acknowledge signal

Call(s):

```
void ioctl(const int *pModuleBase, IIC_TX_ACK,
           UWord16 param);
```

Arguments:

Table 5-658. IIC_TX_ACK ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter specifying whether the next byte received is to be acknowledged. Use one of the predefined constants: IIC_ACK - do acknowledge IIC_NO_ACK - do not acknowledge

Description: Using the *IIC_TX_ACK* ioctl command, the application may disable generating the acknowledge signal after a *data* byte is received (matching address byte is always acknowledged). By disabling an acknowledge signal a receiver informs the transmitting node that it wants to finish the transaction.

This command writes directly the TXAK bit of the CTRL register on MC56F801x or IICC1 on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_TX_ACK* ioctl command is implemented as a macro.

Example 5-578. IIC_TX_ACK

```
ioctl(IIC, IIC_TX_ACK, IIC_NO_ACK)
```

This command disables generating of the acknowledge for the next byte received.

5.17.3.12 IIC_REPEAT_START - generate repeated START condition

Call(s):

```
void ioctl(const int *pModuleBase, IIC_REPEAT_START, NULL);
```

Arguments:

Table 5-659. IIC_REPEAT_START ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_REPEAT_START* ioctl command generates the repeated START condition on the IIC bus, provided the node is active bus Master (otherwise the Arbitration Lost status flag is set).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_REPEAT_START* ioctl command is implemented as a macro.

Example 5-579. IIC_REPEAT_START

```
ioctl(IIC, IIC_REPEAT_START, NULL);
```

This command generates the repeated START condition on the IIC bus.

5.17.3.13 IIC_WRITE_CONTROL_REG - write the Control Register

Call(s):

```
void ioctl(const int *pModuleBase, IIC_WRITE_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-660. IIC_WRITE_CONTROL_REG ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	The value to be written to the CTRL register

Description: The *IIC_WRITE_CONTROL_REG* ioctl command writes a given value directly to the IIC Control Register (CTRL) on MC56F801x or (IICC1) on MC56F800x.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_WRITE_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-580. IIC_WRITE_CONTROL_REG

```
ioctl(IIC, IIC_WRITE_CONTROL_REG, 0);
```

This code writes zero to Control Register, shutting down the IIC module.

5.17.3.14 IIC_READ_CONTROL_REG - read the Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_CONTROL_REG,
              NULL);
```

Arguments:

Table 5-661. IIC_READ_CONTROL_REG ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_CONTROL_REG* ioctl command reads the value of the IIC Control Register (CTRL) on MC56F801x or (IICC1) on MC56F800x.

Returns: Register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-581. IIC_READ_CONTROL_REG

```
UWord16 ctl;

ctl = ioctl(IIC, IIC_READ_CONTROL_REG, NULL);
```

This code reads the content of the Control Register.

5.17.3.15 IIC_READ_DATA - read received data

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_DATA, NULL);
```

Arguments:

Table 5-662. IIC_READ_DATA ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_DATA* ioctl command fetches the byte received from the IIC bus by directly reading the IIC Data I/O Register (DATA). Typically, reading of the byte is done in the IIC interrupt service routine.

Returns: UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_READ_DATA* ioctl command is implemented as a macro.

Example 5-582. IIC_READ_DATA

```
UWord16 b;

b = ioctl(IIC, IIC_READ_DATA, NULL);
```

This code reads the byte received from the IIC.

5.17.3.16 IIC_WRITE_DATA - write data to be transmitted

Call(s):

```
void ioctl(const int *pModuleBase, IIC_WRITE_DATA,
           UWord16 param);
```

Arguments:

Table 5-663. IIC_WRITE_DATA ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Data byte to be transmitted - right aligned in the UWord16 parameter.

Description: The *IIC_WRITE_DATA* ioctl command writes directly to the IIC Data I/O Register (DATA). Typically, writing the data register is done in the IIC interrupt service routine.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_WRITE_DATA* ioctl command is implemented as a macro.

Example 5-583. IIC_WRITE_DATA

```
ioctl(IIC, IIC_WRITE_DATA, 'A');
```

This code writes a data byte to DATA transmit register.

5.17.3.17 IIC_SET_NOISE_FILTER - set noise filter

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_NOISE_FILTER,
           UWord16 param);
```

Arguments:

Table 5-664. IIC_SET_NOISE_FILTER ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Value (0..15) to be written to the Noise Filter Register.

Description: The *IIC_SET_NOISE_FILTER* ioctl command sets the width (in clocks) of the digital filter applied on the SCL and SDA input signals. Any signal transition on any of the two inputs is accepted only if the new state of the input persists longer than the specified number of clocks. Otherwise, it is considered to be a glitch and is ignored by the IIC module.

This command directly writes the NF bit-field of the NFILT register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x.

Design/Implementation: The *IIC_SET_NOISE_FILTER* ioctl command is implemented as a macro.

Example 5-584. IIC_SET_NOISE_FILTER

```
ioctl(IIC, IIC_SET_NOISE_FILTER, 0);
```

This code disables the IIC Noise Filter.

5.17.3.18 IIC_READ_STATUS_REG - read the Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_STATUS_REG, NULL);
```

Arguments:

Table 5-665. IIC_READ_STATUS_REG ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_STATUS_REG* ioctl command reads and returns the value of the IIC Status Register (STAT) on MC56F801x or (IICS) on MC56F800x.

Returns: UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_READ_STATUS_REG* ioctl command is implemented as a macro.

Example 5-585. IIC_READ_STATUS_REG

```
UWord16 status = ioctl(IIC, IIC_READ_STATUS_REG, NULL);
```

This code reads content of the IIC Status Register.

5.17.3.19 IIC_TEST_STATUS_REG - test bits in the Status Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_TEST_STATUS_REG,
          UWord16 param);
```

Arguments:

Table 5-666. IIC_TEST_STATUS_REG ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter selecting the bit or bits to be tested in the Status Register. Use any combination of the following predefined constants:
		IIC_TRANSFER_COMPLETE — Transfer (either Tx or Rx) is complete IIC_ADDRESSED_AS_SLAVE — Set in Slave after addressed by Master IIC_BUS_BUSY — The IIC Bus is currently busy IIC_ARBITRATION_LOST — Arbitration lost (broken Master) IIC_SLAVE_TRANSMIT — Set in Slave in Master-Read transactions IIC_IBUS_INT — IIC Bus Interrupt is pending IIC_RX_ACK — When set, the last transfer was **not** acknowledged

Description: The *IIC_TEST_STATUS_REG* ioctl command reads and tests the selected bit or bits in the IIC Status Register.

Returns: UWord16, non-zero if any of selected bits is non-zero.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_TEST_STATUS_REG* ioctl command is implemented as a macro.

Example 5-586. IIC_TEST_STATUS_REG

```
if(ioctl(IIC, IIC_TEST_STATUS_REG, IIC_ADDRESSED_AS_SLAVE))
{
      // .. this (slave) node was just addressed
      // Now set test the IIC_SLAVE_TRANSMIT and set
      // Transmit or Receive mode accordingly
}
```

This code tests the IAAS (Addressed as slave) bit in the Status Register.

5.17.3.20 IIC_CLEAR_ARBITRATION_LOST - clear Arbitration Lost flag

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_ARBITRATION_LOST,
           NULL);
```

Arguments:

Table 5-667. IIC_CLEAR_ARBITRATION_LOST ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CLEAR_ARBITRATION_LOST* ioctl command clears the "Arbitration Lost" flag in the IIC Status Register. Use this command after you detect and handle the "Arbitration Lost" event, typically in the interrupt service routine.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_CLEAR_ARBITRATION_LOST* ioctl command is implemented as a macro.

Example 5-587. IIC_CLEAR_ARBITRATION_LOST

```
ioctl(IIC, IIC_CLEAR_ARBITRATION_LOST, NULL);
```

This code clears the "Arbitration Lost" status flag.

5.17.3.21 IIC_CLEAR_I_BUS_INT - clear IIC interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_I_BUS_INT,
           NULL);
```

Arguments:

Table 5-668. IIC_CLEAR_I_BUS_INT ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CLEAR_I_BUS_INT* ioctl command clears the interrupt flag in the IIC Status Register. Use this command to acknowledge the IIC interrupt.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F801x and MC56F800x.

Design/Implementation: The *IIC_CLEAR_I_BUS_INT* ioctl command is implemented as a macro.

Example 5-588. IIC_CLEAR_I_BUS_INT

```
ioctl(IIC, IIC_CLEAR_I_BUS_INT, NULL);\
```

This code clears the interrupt flag.

5.17.3.22 IIC_SET_GENERAL_CALL_ADDRESS - enable/disable general call address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_GENERAL_CALL_ADDRESS,
           UWord16 param);
```

Arguments:

Table 5-669. IIC_SET_GENERAL_CALL_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_ENABLE - General Call Address enabled IIC_DISABLE - General Call Address disabled

Description: The *IIC_SET_GENERAL_CALL_ADDRESS* ioctl command enables or disables the General Call Address.

This command writes directly into the GCAEN bit of the Control Register 2 (IICC2).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_GENERAL_CALL_ADDRESS* ioctl command is implemented as a macro.

Example 5-589. IIC_SET_GENERAL_CALL_ADDRESS

```
ioctl(IIC, IIC_SET_GENERAL_CALL_ADDRESS, IIC_ENABLE);
```

This code enables the General Call Address.

5.17.3.23 IIC_SET_ADDRESS_EXTENSION_MODE - select 7-BIT or 10-BITaddress scheme

Call(s):

```
void ioctl(const int *pModuleBase,
           IIC_SET_ADDRESS_EXTENSION_MODE, UWord16 param);
```

Arguments:

Table 5-670. IIC_SET_ADDRESS_EXTENSION_MODE ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_7BIT - 7-bit address mode is selected IIC_10BIT - 10-bit address mode is selected

Description: The *IIC_SET_ADDRESS_EXTENSION_MODE* ioctl command selects 7-bit or 10-bit address scheme.

This command writes directly into the ADEXT bit of the Control Register 2 (IICC2).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_ADDRESS_EXTENSION_MODE* ioctl command is implemented as a macro.

Example 5-590. IIC_SET_ADDRESS_EXTENSION_MODE

```
ioctl(IIC, IIC_SET_ADDRESS_EXTENSION_MODE, IIC_10BIT);
```

This code selects 10-bit address scheme.

5.17.3.24 IIC_SET_ADDRESS_EXTENSION - set upper three bit slave extension address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_ADDRESS_EXTENSION,
           UWord16 param);
```

Arguments:

Table 5-671. IIC_SET_ADDRESS_EXTENSION ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter to set the upper three bits of the slave address in 10-bit address scheme.

Description: The *IIC_SET_ADDRESS_EXTENSION* ioctl command sets the upper three bits of the slave address in the 10-bit address scheme.

This command writes directly into the AD8-AD10 bits of the Control Register 2 (IICC2).

Returns: None.

Range Issues: UWord16 value must be in range of 0 to 7.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_ADDRESS_EXTENSION* ioctl command is implemented as a macro.

Example 5-591. IIC_SET_ADDRESS_EXTENSION

```
ioctl(IIC, IIC_SET_ADDRESS_EXTENSION, (555>>8));
```

This code sets upper three bits of the slave address in the 10-bit address scheme.

5.17.3.25 IIC_SET_10BIT_ADDRESS - set 10-bit slave address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_10BIT_ADDRESS,
           UWord16 param);
```

Arguments:

Table 5-672. IIC_SET_10BIT_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Parameter to set the 10-bit slave address.

Description: The *IIC_SET_10BIT_ADDRESS* ioctl command sets the 10-bit slave address.

This command writes directly into the AD1-AD7 bit-field in the IICA1 register and into the AD8-AD10 bits of the Control Register 2 (IICC2).

Returns: None.

Range Issues: UWord16 value must be in range of 0 to 1023.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_10BIT_ADDRESS* ioctl command is implemented as a macro.

Example 5-592. IIC_SET_10BIT_ADDRESS

```
ioctl(IIC, IIC_SET_10BIT_ADDRESS, 555);
```

This code sets the 10-bit slave address.

5.17.3.26 IIC_GET_10BIT_ADDRESS - read 10-bit slave address

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_GET_10BIT_ADDRESS,
          NULL);
```

Arguments:

Table 5-673. IIC_GET_10BIT_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_GET_10BIT_ADDRESS* ioctl command reads and returns the 10-bit slave address.

This command reads directly from the AD1-AD7 bit-field in the IICA1 register and from the AD8-AD10 bits of the Control Register 2 (IICC2).

Returns: 10-bit slave address.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_GET_10BIT_ADDRESS* ioctl command is implemented as a macro.

Example 5-593. IIC_GET_10BIT_ADDRESS

```
UWord16 address=ioctl(IIC, IIC_GET_10BIT_ADDRESS, NULL);
```

This code reads and returns the 10-bit slave address.

5.17.3.27 IIC_SET_FAST_ACK_NACK - enable/disable fast set up ACK/NACK response

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_FAST_ACK_NACK,
           UWord16 param);
```

Arguments:

Table 5-674. IIC_SET_FAST_ACK_NACK ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_ENABLE - fast set up ACK/NACK response enabled IIC_DISABLE - fast set up ACK/NACK response disabled

Description: The *IIC_SET_FAST_ACK_NACK* ioctl command enables or disables the fast set up ACK/NACK response.

This command writes directly into the FACK bit of the Control and Status Register (IICSMB).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_FAST_ACK_NACK* ioctl command is implemented as a macro.

Example 5-594. IIC_SET_FAST_ACK_NACK

```
ioctl(IIC, IIC_SET_FAST_ACK_NACK, IIC_DISABLE);
```

This code disables the fast set up ACK/NACK response.

5.17.3.28 IIC_SET_SMBUS_RESPONSE_ADDRESS - enable/disable SMBUS alert response address

Call(s):

```
void ioctl(const int *pModuleBase,
           IIC_SET_SMBUS_RESPONSE_ADDRESS, UWord16 param);
```

Arguments:

Table 5-675. IIC_SET_SMBUS_RESPONSE_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_ENABLE - SMBus alert response address enabled IIC_DISABLE - SMBus alert response address disabled

Description: The *IIC_SET_SMBUS_RESPONSE_ADDRESS* ioctl command enables or disables SMBus alert response address.

This command writes directly into the ALERTTEN bit of the Control and Status Register (IICSMB).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_SMBUS_RESPONSE_ADDRESS* ioctl command is implemented as a macro.

Example 5-595. IIC_SET_SMBUS_RESPONSE_ADDRESS

```
ioctl(IIC, IIC_SET_SMBUS_RESPONSE_ADDRESS, IIC_ENABLE);
```

This code ensables SMBus alert response address.

5.17.3.29 IIC_SET_SECOND_IIC_ADDRESS - enable/disable SMBus device default address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_SECOND_IIC_ADDRESS,
           UWord16 param);
```

Arguments:

Table 5-676. IIC_SET_SECOND_IIC_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_ENABLE - SMBus device default address enabled IIC_DISABLE - SMBus device default address disabled

Description: The *IIC_SET_SECOND_IIC_ADDRESS* ioctl command enables or disables SMBus device default address.

This command writes directly into the SIICAEN bit of the Control and Status Register (IICSMB).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_SECOND_IIC_ADDRESS* ioctl command is implemented as a macro.

Example 5-596. IIC_SET_SECOND_IIC_ADDRESS

```
ioctl(IIC, IIC_SET_SECOND_IIC_ADDRESS, IIC_DISABLE);
```

This code disables SMBus device default address.

5.17.3.30 IIC_SET_TIME_OUT_CLOCK - select the clock source of time out counter

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_TIME_OUT_CLOCK,
           UWord16 param);
```

Arguments:

Table 5-677. IIC_SET_TIME_OUT_CLOCK ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the following constants: IIC_DIV1 - sets Time Out Counter clock to bus frequency IIC_DIV64 - sets Time Out Counter clock to bus frequency / 64

Description: The *IIC_SET_TIME_OUT_CLOCK* ioctl command selects the clock of Time Out Counter.

This command writes directly into the TCKSEL bit of the Control and Status Register (IICSMB).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_TIME_OUT_CLOCK* ioctl command is implemented as a macro.

Example 5-597. IIC_SET_TIME_OUT_CLOCK

```
ioctl(IIC, IIC_SET_TIME_OUT_CLOCK, IIC_DIV1);
```

This code select Time Out Counter clock to bus frequency.

5.17.3.31 IIC_TEST_TIMEOUT_FLAGS - tests selected timeout flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_TEST_TIMEOUT_FLAGS,
          UWord16 param);
```

Arguments:

Table 5-678. IIC_TEST_TIMEOUT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use combination of the predefined constants: IIC_LOW_TIMEOUT |... to test low time out flag IIC_HIGH_TIMEOUT ... to test high time out flag

Description: The *IIC_TEST_TIMEOUT_FLAGS* ioctl command returns the value of the selected time out flags.

This command returns the SLTF or SHTF bits of the Control and Status Register (IICSMB).

Returns: UWord16 value of the selected interrupt status bits.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_TEST_TIMEOUT_FLAGS* command is implemented as a macro.

Example 5-598. IIC_TEST_TIMEOUT_FLAGS

```
if(ioctl(IIC, IIC_TEST_TIMEOUT_FLAGS, IIC_LOW_TIMEOUT))
{
      ...
}
```

This code tests low timeout flag of the IIC module.

5.17.3.32 IIC_CLEAR_LOW_TIMEOUT_FLAG - clear low timeout flag

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_LOW_TIMEOUT_FLAG,
           NULL);
```

Arguments:

Table 5-679. IIC_CLEAR_LOW_TIMEOUT_FLAG ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CLEAR_LOW_TIMEOUT_FLAG* ioctl command clears low timeout flag by writing 1 to it.

This command writes 1 into the SLTF bit of the Control and Status Register (IICSMB).

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_CLEAR_LOW_TIMEOUT_FLAG* command is implemented as a macro.

Example 5-599. IIC_CLEAR_LOW_TIMEOUT_FLAG

```
if(ioctl(IIC, IIC_TEST_TIMEOUT_FLAGS, IIC_LOW_TIMEOUT))
{
      ...
}

ioctl(IIC, IIC_CLEAR_LOW_TIMEOUT_FLAG, NULL);
```

This code shows how the low timeout flag is read, processed and cleared.

5.17.3.33 IIC_SET_SMBUS_ADDRESS - set the IIC slave SMBus address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_SMBUS_ADDRESS,
           UWord16 param);
```

Arguments:

Table 5-680. IIC_SET_SMBUS_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	The 7-bit address (0...127) to be used in the slave mode by the SMBus.

Description: The *IIC_SET_SMBUS_ADDRESS* ioctl command can be used when changing the IIC SMBus node address.

This command writes directly into the SAD bit-field in the I2C_ADDR.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_SET_SMBUS_ADDRESS* ioctl command is implemented as a macro.

Example 5-600. IIC_SET_SMBUS_ADDRESS

```
ioctl(IIC, IIC_SET_SMBUS_ADDRESS, 100);
```

This code sets the IIC SMBus slave address to 100.

5.17.3.34 IIC_GET_SMBUS_ADDRESS - get the currently active slave address

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_GET_SMBUS_ADDRESS,
              NULL);
```

Arguments:

Table 5-681. IIC_GET_SMBUS_ADDRESS ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_SET_SMBUS_ADDRESS* ioctl command reads and returns the IIC SMBus slave address.

Returns: The IIC address of the node in range 0..127 as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_GET_SMBUS_ADDRESS* ioctl command is implemented as a macro.

Example 5-601. IIC_GET_SMBUS_ADDRESS

```
UWord16 a;

a = ioctl(IIC, IIC_GET_SMBUS_ADDRESS, NULL);
```

This code reads the actual SMBus slave address.

5.17.3.35 IIC_WRITE_SCL_LOW_TIMEOUT - sets SCL low timeout value

Call(s):

```
void ioctl(const int *pModuleBase, IIC_WRITE_SCL_LOW_TIMEOUT,
           UWord16 param);
```

Arguments:

Table 5-682. IIC_WRITE_SCL_LOW_TIMEOUT ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.
param	in	The 15-bit SCL low timeout value (0...65535).

Description: The *IIC_WRITE_SCL_LOW_TIMEOUT* ioctl command sets the SCL low timeout.

This command writes directly into the SSLT bit-fields in the I2C_SLT1 register and in the I2C_SLT2 register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_WRITE_SCL_LOW_TIMEOUT* ioctl command is implemented as a macro.

Example 5-602. IIC_WRITE_SCL_LOW_TIMEOUT

```
ioctl(IIC, IIC_SET_TIME_OUT_CLOCK, IIC_DIV64);
ioctl(IIC, IIC_WRITE_SCL_LOW_TIMEOUT, 10000);
```

This code sets the SCL low timeout to 640000 bus clocks.

5.17.3.36 IIC_READ_SCL_LOW_TIMEOUT - read SCL low timeout value

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_SCL_LOW_TIMEOUT,
              NULL);
```

Arguments:

Table 5-683. IIC_READ_SCL_LOW_TIMEOUT ioctl call arguments

pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_SET_SMBUS_ADDRESS* ioctl command reads and returns the SCL low timeout value.

Returns: The IIC SCL low timeout value in range 0..65535 as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *IIC_READ_SCL_LOW_TIMEOUT* ioctl command is implemented as a macro.

Example 5-603. IIC_READ_SCL_LOW_TIMEOUT

```
UWord16 a;

a = ioctl(IIC, IIC_READ_SCL_LOW_TIMEOUT, NULL);
```

This code reads the SCL low timeout value.

5.17.4 IIC Driver Sample Application

The IIC driver application is designed for Freescale EVMs and is intended to illustrate the usage of this driver by a real example. For the IIC module, the sample application contains a “complete” IIC interrupt handler code, implementing the state diagram described below in Figure 5-3. The code can be reused in user applications as a general-purpose IIC communication driver.

The IIC sample application demonstrates both the Master and Slave capabilities of the IIC peripheral module. You will need two MC56F8013DEMO boards connected together via the IIC bus (3 wires needed: SCL, SDA and GROUND).

1. Most of the time, the IIC is in Slave mode and waits for being addressed by the Master device.
   - a) If Master uses a WRITE transaction, the BIT_0 and BIT_1 of (each) data byte received are tested and we toggle the YELLOW[GPIO_A1] and/or GREEN[GPIO_A2] LEDs if BIT_0/1 is set.
   - b) If Master uses a READ transaction, we respond with a byte where BIT_0 and BIT_1 report what LEDs (GPIO_A1 and GPIO_A2) are ON.
2. Using an GPIO interrupt, we monitor the board buttons. Any time the button is pressed, the IIC is switched to Master mode and a message is sent to a remote slave. The BIT_0 and BIT_1 bits are set in the message data byte if button #1 and/or #2 were pressed. Thanks to (1a) the remote node should respond by toggling the YELLOW and/or GREEN LEDs.
3. Even if no button is pressed, the IIC is periodically (approx. once per second) switched to a Master mode and the remote note is queried for what LEDs are currently turned-on (see 1b). The returned message is then used to turn the status YELLOW and GREEN LEDs on or off (GPIO_A4 and GPIO_A5).

NOTE:

The IIC device must not address itself so it is required to define different REMOTE_ADDRES and the local slave address (in graphical configuration tool).

NOTE:

The Graphical Configuration Tool was used to configure CPU pins and peripheral modules in this application. See the *appconfig.h* file listing for more details.

The application can be found at e.g. *sample_applications\MC56F8013DEMO\iic_demo* directory and consists of the application project *iic_demo.mcp*, source code for the application *main.c* and configuration file *appconfig.h*.

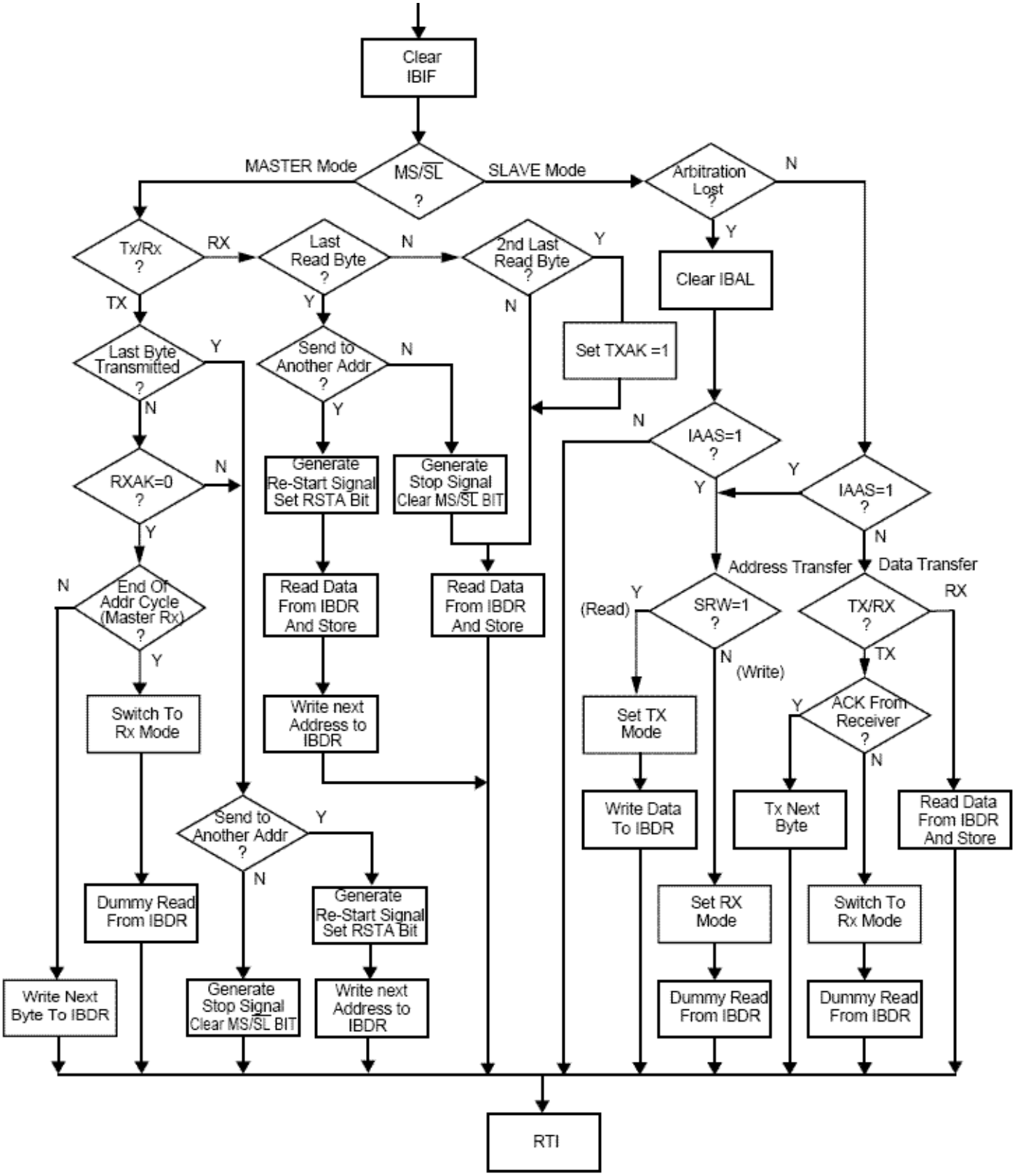


Figure 5-3. IIC Interrupt Handling

5.18 IIC Driver (MC56F802x/3x)

This section describes the DSP56800E_Quick_Start API for the Inter-Integrated Circuit (IIC) Bus Controller which is implemented on MC56F802x/3x family of microcontrollers. The functionality of the IIC module itself is described in the **56F802x/3x Peripheral Reference Manual**.

5.18.1 Introduction

The Inter-IC Bus (IIC or I2C) is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange between devices. The MC56F802x/3x family of microcontrollers implements the new IIC module comparing to previous implementations on MC56F801x or other families like HC08 or HC12. The main features of the new module are the following:

- Two-wire I2C serial interface consists of a serial data (SDA) line and a serial clock (SCL) line
- Two speed modes:
  - — Standard mode
  - — Fast mode
- Clock synchronization
- Master or slave I2C operation
- Supports multi-master operation (bus arbitration)
- 7- or 10-bit addressing
- 7- or 10-bit combined format transfers
- Slave bulk transmit mode
- Ignores CBUS addresses (an older ancestor of I2C previously shared the I2C bus)
- Four transmit and four receive buffers
- Interrupt or polled-mode operation
- Handles bit and byte waiting at all bus speeds
- Digital filter for the received SDA and SCL lines

This section describes the IIC driver software, providing the low level software layer interfacing hardware with software.

5.18.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.18.3.

Table 5-684. IIC Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F83xx
IIC (IIC_BASE)	N/A (different module)	0xF280	N/A

5.18.2.1 API Definition

The following header files are needed in order to use the IIC device driver:

Required Header File(s):

```
#include "qs.h"
#include "iic.h"
```

The following information may be found in the header file *iic.h*.

Public Data Structure(s):

None

5.18.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static IIC module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-685. IIC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
IIC_ENBL_INIT	UWord16	Initial value of the IIC Enable Register
IIC_CTRL_INIT	UWord16	Initial value of the IIC Control Register
IIC_SAR_INIT	UWord16	Initial value of the IIC Slave Address Register
IIC_RXFT_INIT	UWord16	Initial value of the IIC Receive FIFO Threshold Register
IIC_TXFT_INIT	UWord16	Initial value of the IIC Transmit FIFO Threshold Register
IIC_SSHCNT_INIT	UWord16	Initial value of the IIC Standard Speed Clock High Count Register
IIC_SSLCNT_INIT	UWord16	Initial value of the IIC Standard Speed Clock Low Count Register
IIC_FSHCNT_INIT	UWord16	Initial value of the IIC Fast Speed Clock High Count Register
IIC_FSLCNT_INIT	UWord16	Initial value of the IIC Fast Speed Clock Low Count Register
IIC_IENBL_INIT	UWord16	Initial value of the IIC Interrupt Enable Register

5.18.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the IIC device modes or accesses the IIC register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-686. IIC Driver Arguments - ioctl

pModuleBase	in	IIC module identifier. Use IIC.
cmd	in	Commands found in *iic.h* which are used to modify the IIC module status and control registers. See Table 5-687.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-687. ioctl commands

Cmd	param	Return	Description
IIC_INIT	NULL	None	Initializes IIC module by data from configuration file (appconfig.h).
IIC_MODULE	IIC_ENABLE/ IIC_DISABLE	None	Enables or disables IIC device.

Table 5-687. ioctl commands (Continued)

Cmd	param	Return	Description
IIC_TRY_GRACEFUL_SHUTDOWN	UWord16 (as timeout in 16x core clocks)	UWord16 code	Tries to perform graceful disconnect from the IIC bus and de-activation of IIC module. Returns one of status codes: IIC_SHUTDOWN_SUCCESS IIC_SHUTDOWN_STILLBUSY IIC_SHUTDOWN_REACTIVATED
Initialization time-only commands:			
IIC_SLAVE_OPERATION	IIC_ENABLE/ IIC_DISABLE	None	Enables or disables slave operation.
IIC_MASTER_OPERATION	IIC_ENABLE/ IIC_DISABLE	None	Enables or disables master mode operation.
IIC_USE_REPEATED_START	IIC_ENABLE/ IIC_DISABLE	None	Enables or disables repeated-start condition to be generated in master mode.
IIC_SET_ADDRESS	UWord16 (Address value. When 10-bit address is to be specified, combine the address value with: IIC_10BIT_ADDRESS)	None	Sets slave address.
IIC_SET_SPEED_MODE	IIC_NORMAL_SPEED / IIC_FAST_SPEED	None	Sets speed mode for master mode operation.
IIC_SET_SSHCNT	UWord16	None	Sets the Standard-Speed SCL-High time in clock counts.
IIC_SET_SSLCNT	UWord16	None	Sets the Standard-Speed SCL-Low time in clock counts.
IIC_SET_FSHCNT	UWord16	None	Sets the Fast-Speed SCL-High time in clock counts.
IIC_SET_FSLCNT	UWord16	None	Sets the Fast-Speed SCL-Low time in clock counts.
Master-initiated transactions:			
IIC_CAN_INITIATE_TRANSACTION	NULL	UWord16	Returns non-zero value when master transaction may be started.
IIC_INITIATE_TRANSACTION	UWord16 (Address value. When 10-bit address is to be specified, combine the address value with: IIC_10BIT_ADDRESS)	None	Initiates standard transaction to the addressed target.

Table 5-687. ioctl commands (Continued)

Cmd	param	Return	Description
IIC_INITIATE_SB_TRANSACTION	UWord16 (Address value. When 10-bit address is to be specified, combine the address value with: IIC_10BIT_ADDRESS)	None	Initiates “start-byte” transaction to the addressed target.
IIC_INITIATE_GC_TRANSACTION	NULL	None	Initiates “general-call” transaction (target address 0).
IIC_MASTER_TRANSACTION_ACTIVE	NULL	UWord16	Returns non-zero when master state machine is active.
Data I/O			
IIC_CAN_READ_DATA	NULL	UWord16	Returns non-zero when receiver FIFO contains at least one byte to be read.
IIC_READ_DATA	NULL	UWord16	Reads one byte from receiver FIFO.
IIC_CAN_WRITE_DATA	NULL	UWord16	Returns non-zero when there is a space left in transmitter FIFO to accept at least one byte.
IIC_WRITE_DATA	UWord16 (one byte data)	None	Writes one byte to transmit FIFO.
IIC_CAN_REQUEST_DATA	NULL	UWord16	Returns non-zero when there is a space left in receiver FIFO to accept at least one byte.
IIC_READ_REQUEST	NULL	None	Requests to clock-in one byte as a part of master-receive operation.
Interrupts:			
IIC_INT_ENABLE IIC_INT_DISABLE	Any combination of IIC_TXABRT_INT | IIC_TXOVR_INT | IIC_RXOVR_INT | IIC_RXUND_INT | IIC_GC_INT | IIC_RXFULL_INT | IIC_TXDONE_INT | IIC_RDREQ_INT | IIC_TXEMPTY_INT | IIC_STDET_INT | IIC_STPDET_INT | IIC_ACT_INT	None	Enables or disables selected IIC interrupts. There are five interrupt vectors dividing interrupts into five categories: Error, General Call, Receiver, Transmitter, Status.
IIC_READ_INT_FLAGS	NULL	UWord16	Reads all interrupt flags.

Table 5-687. ioctl commands (Continued)

Cmd	param	Return	Description
IIC_TEST_INT_FLAGS	Any combination of IIC_TXABRT_INT | IIC_TXOVR_INT | IIC_RXOVR_INT | IIC_RXUND_INT | IIC_GC_INT | IIC_RXFULL_INT | IIC_TXDONE_INT | IIC_RDREQ_INT | IIC_TXEMPTY_INT | IIC_STDET_INT | IIC_STPDET_INT | IIC_ACT_INT	UWord16	Returns non-zero if any of the selected interrupt flags are set.
IIC_CLEAR_EINT_FLAGS	Any combination of interrupt flags may be supplied. Only the following error flags are evaluated: IIC_TXABRT_INT | IIC_TXOVR_INT | IIC_RXOVR_INT | IIC_RXUND_INT	None	Clears error-interrupt flags selected by the parameter value.
IIC_CLEAR_GINT_FLAGS	Any combination of interrupt flags may be supplied. Only the following error flag is evaluated: IIC_GC_INT	None	Clears general-call interrupt flag if this bit is selected by parameter value.
IIC_CLEAR_TINT_FLAGS	Any combination of interrupt flags may be supplied. Only the following error flags are evaluated: IIC_TXDONE_INT | IIC_RDREQ_INT	None	Clears transmitter-interrupt flags selected by the parameter value. The IIC_TXEMPTY_INT interrupt flag can only be cleared by writing data into TX FIFO.
IIC_CLEAR_SINT_FLAGS	Any combination of interrupt flags may be supplied. Only the following error flags are evaluated: IIC_STDET_INT | IIC_STPDET_INT | IIC_ACT_INT	None	Clears status-interrupt flags selected by the parameter value.
IIC_CLEAR_ALL_INTS	NULL	None	Clears all interrupt flags and the TXABRTSRC register.
IIC_READ_TXABORT_SOURCE	NULL	UWord16	Returns value of the TXABRTSRC Register.

Table 5-687. ioctl commands (Continued)

Cmd	param	Return	Description
IIC_CLEAR_TXABORT_SOURCE	NULL	NULL	Clears TXABRT interrupt flag and the TXABRTSRC Register.
Status:			
IIC_READ_STATUS_FLAGS	NULL	UWord16	Returns value of the Status Register.
IIC_TEST_STATUS_FLAGS	Any combination of IIC_STAT_xxx bits	UWord16	Returns non-zero if any of given Status Register bits is set.

5.18.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.18.3.1 IIC_INIT - initialize the IIC module

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INIT, NULL);
```

Arguments:

Table 5-688. IIC_INIT ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_INIT* ioctl command executes the initialization function, which initializes all configured IIC registers by the values defined in *appconfig.h*. It is intended for static configuration of the IIC module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INIT* ioctl command is implemented as a function call. Only necessary IIC registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-604. IIC_INIT

```
ioctl(IIC, IIC_INIT, NULL);
```

This code initializes the IIC module by the values defined in the *appconfig.h* file. The *appconfig.h* file can be edited manually or generated by the Graphical Configuration Tool.

5.18.3.2 IIC_MODULE - enable/disable IIC device

Call(s):

```
void ioctl(const int *pModuleBase, IIC_MODULE, UWord16 param);
```

Arguments:

Table 5-689. IIC_MODULE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use IIC_ENABLE to enable or IIC_DISABLE to disable IIC module.

Description: The *IIC_MODULE* ioctl command enables (IIC_ENABLE) or disables (IIC_DISABLE) the IIC module. Before the IIC module is used, the IIC device should be configured (in disabled state) and then enabled. Also it may be decided to let the IIC module to be enabled already during the initialization procedure.

A care should be taken when hard-disabling the IIC module using this command as it may cause IIC Bus errors. The IIC_TRY_GRACEFUL_SHUTDOWN command should be used to safely disable the IIC module.

This command writes directly into the Enable (EN) bit of the IIC Enable Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_MODULE* command is implemented as a macro.

Example 5-605. IIC_MODULE

```
/* enable it */
ioctl(IIC, IIC_MODULE, IIC_ENABLE);
```

This code enables the IIC module.

5.18.3.3 IIC_TRY_GRACEFUL_SHUTDOWN - safely disable the IIC module

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_TRY_GRACEFUL_SHUTDOWN,
     UWord16 param);
```

Arguments:

Table 5-690. IIC_TRY_GRACEFUL_SHUTDOWN ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16 value specifying the timeout when waiting for IIC idle condition. The timeout value is specified in 16x core clock units. Value of 0 specifies maximum timeout value (16x65536) core clocks.

Description: The *IIC_TRY_GRACEFUL_SHUTDOWN* ioctl command disables the IIC module. Comparing to IIC_MODULE command with the IIC_DISABLE parameter, this command performs activity status polling to detect when it is safe to disable the module. The parameter value gives the time-out period for the polling loop and the command returns one of two error codes when time-out expires. The IIC module is disabled only if this command reports success code.

See the **56F802x/3x Peripheral Reference Manual** for more details about the IIC shutdown procedure.

Returns: One of the result codes as UWord16:

- IIC_SHUTDOWN_SUCCESS ... IIC module successfully disabled
- IIC_SHUTDOWN_STILLBUSY ... Timeout when waiting for idle condition.
- IIC_SHUTDOWN_REACTIVATED ... Activity detected on the bus within the “time-out” period after the IIC module was disabled. Module was reactivated not to cause IIC bus error.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_TRY_GRACEFUL_SHUTDOWN* command is implemented as a function call.

Example 5-606. IIC_TRY_GRACEFUL_SHUTDOWN

```
if(ioctl(IIC,IIC_TRY_GRACEFUL_SHUTDOWN,0)==IIC_SHUTDOWN_SUCCESS)
{
     // IIC disabled
}
else
{
     // IIC still active
}
```

This code performs the graceful shutdown of the IIC module with a maximum possible time-out value.

5.18.3.4 IIC_SLAVE_OPERATION - enable or disable slave operation

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SLAVE_OPERATION,
     UWord16 param);
```

Arguments:

Table 5-691. IIC_SLAVE_OPERATION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use IIC_ENABLE to enable or IIC_DISABLE to disable the slave operation of the IIC module.

Description: The *IIC_SLAVE_OPERATION* ioctl command enables or disables the slave operation of the IIC module. This is a configuration time-only command and has no effect when IIC module is enabled.

This command writes to the Slave Disable (SLVDIS) bit of the IIC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x. This command may only be applied when IIC module is disabled.

Design/Implementation: The *IIC_SLAVE_OPERATION* command is implemented as a macro.

Example 5-607. IIC_SLAVE_OPERATION

```
ioctl(IIC, IIC_SLAVE_OPERATION, IIC_ENABLE);
```

This code enables the slave operation if the IIC module.

5.18.3.5 IIC_MASTER_OPERATION - enable or disable master operation

Call(s):

```
void ioctl(const int *pModuleBase, IIC_MASTER_OPERATION,
     UWord16 param);
```

Arguments:

Table 5-692. IIC_MASTER_OPERATION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use IIC_ENABLE to enable or IIC_DISABLE to disable the master operation of the IIC module.

Description: The *IIC_MASTER_OPERATION* ioctl command enables or disables the master operation of the IIC module. This is a configuration time-only command and has no effect when IIC module is enabled.

This command writes to the Slave Disable (SLVDIS) bit of the IIC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x. This command may only be applied when IIC module is disabled.

Design/Implementation: The *IIC_MASTER_OPERATION* command is implemented as a macro.

Example 5-608. IIC_MASTER_OPERATION

```
ioctl(IIC, IIC_MASTER_OPERATION, IIC_ENABLE);
```

This code enables the master operation if the IIC module.

5.18.3.6 IIC_USE_REPEATED_START - enable or disable repeated start condition

Call(s):

```
void ioctl(const int *pModuleBase, IIC_USE_REPEATED_START,
     UWord16 param);
```

Arguments:

Table 5-693. IIC_USE_REPEATED_START ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use IIC_ENABLE to enable or IIC_DISABLE to disable the repeated start condition.

Description: The *IIC_USE_REPEATED_START* ioctl command enables or disables generating of repeated-start condition on the IIC bus when IIC module operates in a master mode. The repeated start condition has to be disabled when some older slaves are attached to the bus and do not support handling this signal.

This is a configuration time-only command and has no effect when IIC module is enabled.

This command writes to the Repeated Start Enable (RSTEN) bit of the IIC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x. This command may only be applied when IIC module is disabled.

Design/Implementation: The *IIC_USE_REPEATED_START* command is implemented as a macro.

Example 5-609. IIC_USE_REPEATED_START

```
ioctl(IIC, IIC_USE_REPEATED_START, IIC_ENABLE);
```

This code enables the repeated start signal to be generated by the IIC module when operating in master mode.

5.18.3.7 IIC_SET_ADDRESS - set slave address

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_ADDRESS,
     UWord16 param);
```

Arguments:

Table 5-694. IIC_SET_ADDRESS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16 slave address. A 7-bit value is accepted by default. When 10-bit address is to be specified, combine the address value with the IIC_10BIT_ADDRESS flag.

Description: The *IIC_SET_ADDRESS* ioctl command configures the address of the IIC node for a slave-mode operation. Standard 7-bit or extended 10-bit address value may be specified.

This is a configuration time-only command and has no effect when IIC module is enabled.

This command writes directly to the Slave Address Register (SAR).

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x. This command may only be applied when IIC module is disabled.

Design/Implementation: The *IIC_SET_ADDRESS* command is implemented as a macro.

Example 5-610. IIC_SET_ADDRESS

```
ioctl(IIC, IIC_SET_ADDRESS, 0x10);
```

This code configures the 7-bit slave address of the IIC module.

5.18.3.8 IIC_SET_SPEED_MODE - set speed mode for master operation

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_SPEED_MODE,
     UWord16 param);
```

Arguments:

Table 5-695. IIC_SET_SPEED_MODE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Use one of the predefined constants: IIC_NORMAL_SPEED ... standard speed mode (up to 100 kbps) IIC_FAST_SPEED ... fast speed mode (up to 400 kbps)

Description: The *IIC_SET_SPEED_MODE* ioctl command selects the speed mode which is applied when operating in the IIC master mode. This is a configuration time-only command and has no effect when IIC module is enabled.

This command writes to the Speed (SPD) bit-field of the IIC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x. This command may only be applied when IIC module is disabled.

Design/Implementation: The *IIC_SET_SPEED_MODE* command is implemented as a macro.

Example 5-611. IIC_SET_SPEED_MODE

```
ioctl(IIC, IIC_SET_SPEED_MODE, IIC_FAST_SPEED);
```

This code selects the fast speed mode.

5.18.3.9 IIC_SET_*xxx*CNT - set IIC clock timing parameters

Call(s):

```
void ioctl(const int *pModuleBase, IIC_SET_SSHCNT, UWord16 param);
void ioctl(const int *pModuleBase, IIC_SET_SSLCNT, UWord16 param);
void ioctl(const int *pModuleBase, IIC_SET_FSHCNT, UWord16 param);
void ioctl(const int *pModuleBase, IIC_SET_FSLCNT, UWord16 param);
```

Arguments:

Table 5-696. IIC_SET_*xxx*CNT ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16 as the number of IIC clocks (typically IP Bus Clocks).

Description: The *IIC_SET_xxxCNT* ioctl commands write a modified parameter value to an appropriate clock-count register. These values affect how long (in IIC clocks, which are typically IP bus clocks) is the high and low portion of the IIC SCL bus clock generated. The parameter value is modified so that it may specify directly the number of IIC module clocks instead of a raw value required to be written into the register (see the list below). The clock is generated only when IIC operates in master mode.

There are four commands to access these registers, substitute xxxCNT letters in the command name for:

- **SSHCNT** ... SCL high time when standard-speed mode is selected. The SSHCNT Register is written the parameter value less 8 as required. A valid range for the parameter value is 14..65535.
- **SSLCNT** ... SCL low time when standard-speed mode is selected. The SSLCNT Register is written the parameter value less 1 as required. A valid range for the parameter value is 9..65535.
- **FSHCNT** ... SCL high time when fast-speed mode is selected. The FSHCNT Register is written the parameter value less 8 as required. A valid range for the parameter value is 14..65535.
- **FSLCNT** ... SCL low time when fast-speed mode is selected. The FSLCNT Register is written the parameter value less 1 as required. A valid range for the parameter value is 9..65535.

This is a configuration time-only command and has no effect when IIC module is enabled.

Returns: None.

Range Issues: See the bullet list above.

Special Issues: These commands are applicable only on MC56F802x/3x. These commands may only be applied when IIC module is disabled.

Design/Implementation: All *IIC_SET_xxxCNT* commands are implemented as macros.

Example 5-612. IIC_SET_*xxx*CNT

```
ioctl(IIC, IIC_SET_FSHCNT, 43);
ioctl(IIC, IIC_SET_FSLCNT, 37);
```

This code selects the SCL high and low times for the fast speed mode. In result, the 400 kbps speed is selected when IIC operates at 32 MHz.

5.18.3.10 IIC_CAN_INITIATE_TRANSACTION - check if a new master transaction can be initiated

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
      IIC_CAN_INITIATE_TRANSACTION, NULL);
```

Arguments:

Table 5-697. IIC_CAN_INITIATE_TRANSACTION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CAN_INITIATE_TRANSACTION* ioctl command tests the MSTACT and TFE bits in the IIC Status Register and returns a non-zero value when the Master state-machine is idle and TX FIFO is empty. This is the condition which needs to be satisfied before a new master-transaction may be initiated.

Returns: UWord16 to be used in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CAN_INITIATE_TRANSACTION* command is implemented as a macro.

Example 5-613. IIC_CAN_INITIATE_TRANSACTION

```
if(ioctl(IIC, IIC_CAN_INITIATE_TRANSACTION, NULL))
{
      // initiate master transaction to a target address 0x10
      ioctl(IIC, IIC_INITIATE_TRANSACTION, 0x10);
}
```

This code checks if a new master transaction can be initiated.

5.18.3.11 IIC_INITIATE_TRANSACTION - initiate master transaction

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INITIATE_TRANSACTION,
     UWord16 param);
```

Arguments:

Table 5-698. IIC_INITIATE_TRANSACTION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16 target address. A 7-bit value is accepted by default. When 10-bit address is to be specified, combine the address value with the IIC_10BIT_ADDRESS flag.

Description: The *IIC_INITIATE_TRANSACTION* ioctl command begins a new IIC transaction to a selected target. The target address is specified as a parameter, either a standard 7-bit value or an extended 10-bit value (value combined with IIC_10BIT_ADDRESS flag). After the transaction is started, the data may be written (see IIC_WRITE_DATA) or requested and then subsequently read (see IIC_READ_REQUEST and IIC_READ_DATA).

This command writes the Target Address Register (TAR) with SPCL and GCSTRT bits cleared.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INITIATE_TRANSACTION* command is implemented as a macro.

Example 5-614. IIC_INITIATE_TRANSACTION

```
if(ioctl(IIC, IIC_CAN_INITIATE_TRANSACTION, NULL))
{
     // initiate master transaction to a target address 0x10
     ioctl(IIC, IIC_INITIATE_TRANSACTION, 0x10);
}
```

This code checks if a new master transaction can be initiated and, if possible, it initiates the transaction to a node 0x10.

5.18.3.12 IIC_INITIATE_SB_TRANSACTION - initiate “start byte” master transaction

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INITIATE_SB_TRANSACTION,
     UWord16 param);
```

Arguments:

Table 5-699. IIC_INITIATE_SB_TRANSACTION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16 target address. A 7-bit value is accepted by default. When 10-bit address is to be specified, combine the address value with IIC_10BIT_ADDRESS flag.

Description: The *IIC_INITIATE_SB_TRANSACTION* ioctl command begins a new IIC “start byte” transaction to a selected target. The difference between standard and “start byte” transaction is that a special byte value of seven zeros followed by one is inserted to a beginning of a transaction in the “start byte” mode. This enables the bus-polling slave nodes without a dedicated IIC hardware to under-sample the address phase and switch to full sampling rate only after a zero is detected on data line.

The target address is specified as a parameter, either a standard 7-bit value or an extended 10-bit value (value combined with IIC_10BIT_ADDRESS flag). After the transaction is started, the data may be written (see IIC_WRITE_DATA) or requested and then subsequently read (see IIC_READ_REQUEST and IIC_READ_DATA).

This command writes the Target Address Register (TAR) with SPCL and GCSTRT bits set.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INITIATE_SB_TRANSACTION* command is implemented as a macro.

Example 5-615. IIC_INITIATE_SB_TRANSACTION

```
if(ioctl(IIC, IIC_CAN_INITIATE_TRANSACTION, NULL))
{
     // initiate master transaction to a target address 0x10
     ioctl(IIC, IIC_INITIATE_SB_TRANSACTION, 0x10);
}
```

This code checks if a new master transaction can be initiated and, if possible, it initiates the “start byte” transaction to a node 0x10.

5.18.3.13 IIC_INITIATE_GC_TRANSACTION - initiate “general call” master transaction

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INITIATE_GC_TRANSACTION,
     NULL);
```

Arguments:

Table 5-700. IIC_INITIATE_GC_TRANSACTION ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_INITIATE_GC_TRANSACTION* ioctl command begins a new IIC “general call” transaction. General call is transmit-only transaction to address of zero and may be used to broadcast a message to all slave nodes which are capable of handling such “General” calls.

After the transaction is started, the data may only be written using the IIC_WRITE_DATA command.

This command writes the Target Address Register (TAR) with SPCL bit set and GCSTRT bit cleared.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INITIATE_GC_TRANSACTION* command is implemented as a macro.

Example 5-616. IIC_INITIATE_GC_TRANSACTION

```
if(ioctl(IIC, IIC_CAN_INITIATE_TRANSACTION, NULL))
{
     // initiate master transaction to a target address 0x10
     ioctl(IIC, IIC_INITIATE_GC_TRANSACTION, NULL);
}
```

This code checks if a new master transaction can be initiated and, if possible, it initiates the “general call” transaction.

5.18.3.14 IIC_MASTER_TRANSACTION_ACTIVE - check if master transaction is active

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
      IIC_MASTER_TRANSACTION_ACTIVE, NULL);
```

Arguments:

Table 5-701. IIC_MASTER_TRANSACTION_ACTIVE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_MASTER_TRANSACTION_ACTIVE* ioctl command tests if the Master-mode state machine is active or idle. This command returns a non-zero value when state machine is busy, which may be interpreted as that master-initiated transaction is still in progress.

This command tests the MSTACT bit of the IIC Status Register.

Returns: UWord16 to be used in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_MASTER_TRANSACTION_ACTIVE* command is implemented as a macro.

Example 5-617. IIC_MASTER_TRANSACTION_ACTIVE

```
if(ioctl(IIC, IIC_MASTER_TRANSACTION_ACTIVE, NULL))
{
      // master transaction is in progress
}
```

This code checks if a master transaction is in progress.

5.18.3.15 IIC_CAN_READ_DATA - check if data received

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_CAN_READ_DATA,
      NULL);
```

Arguments:

Table 5-702. IIC_CAN_READ_DATA ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CAN_READ_DATA* ioctl command returns non-zero if there is at least one byte in the receive FIFO. If true, the data byte may be retrieved with the IIC_READ_DATA command.

This command tests the RFNE bit of the IIC Status Register.

Returns: UWord16 to be used in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CAN_READ_DATA* command is implemented as a macro.

Example 5-618. IIC_CAN_READ_DATA

```
if(ioctl(IIC, IIC_CAN_READ_DATA, NULL))
{
      data = ioctl(IIC, IIC_READ_DATA, NULL);
}
```

This code checks if a receive byte is available and reads it.

5.18.3.16 IIC_READ_DATA - read received data byte

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_DATA, NULL);
```

Arguments:

Table 5-703. IIC_READ_DATA ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_DATA* ioctl command reads and returns a single data byte from the receive FIFO.

This command reads the lower byte of the IIC Data Register.

Returns: Data byte read as UWord16.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_READ_DATA* command is implemented as a macro.

Example 5-619. IIC_READ_DATA

```
if(ioctl(IIC, IIC_CAN_READ_DATA, NULL))
{
      data = ioctl(IIC, IIC_READ_DATA, NULL);
}
```

This code checks if a receive byte is available and reads it.

5.18.3.17 IIC_CAN_WRITE_DATA - check if data can be transmitted

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_CAN_WRITE_DATA,
      NULL);
```

Arguments:

Table 5-704. IIC_CAN_WRITE_DATA ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CAN_WRITE_DATA* ioctl command returns non-zero if there is at least one free space left in the transmit FIFO.

This command tests the TFNF bit of the IIC Status Register.

Returns: UWord16 to be used in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CAN_WRITE_DATA* command is implemented as a macro.

Example 5-620. IIC_CAN_WRITE_DATA

```
if(ioctl(IIC, IIC_CAN_WRITE_DATA, NULL))
{
      ioctl(IIC, IIC_WRITE_DATA, data);
}
```

This code checks if the IIC transmitter accepts another byte to its FIFO and, if yes, it writes the data byte for transmission.

5.18.3.18 IIC_WRITE_DATA - transmit data byte

Call(s):

```
void ioctl(const int *pModuleBase, IIC_WRITE_DATA,
     UWord16 param);
```

Arguments:

Table 5-705. IIC_WRITE_DATA ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16. The lower byte is written to the transmit FIFO. Upper byte is ignored.

Description: The *IIC_WRITE_DATA* ioctl command writes a given data byte into the transmit FIFO.

This command writes the value into the IIC Data Register, the CMD bit is kept cleared.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_WRITE_DATA* command is implemented as a macro.

Example 5-621. IIC_WRITE_DATA

```
if(ioctl(IIC, IIC_CAN_WRITE_DATA, NULL))
{
     ioctl(IIC, IIC_WRITE_DATA, data);
}
```

This code checks if the IIC transmitter accepts another byte to its FIFO and, if yes, it writes the data byte for transmission.

5.18.3.19 IIC_CAN_REQUEST_DATA - check if data can be requested

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_CAN_REQUEST_DATA,
      NULL);
```

Arguments:

Table 5-706. IIC_CAN_REQUEST_DATA ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CAN_REQUEST_DATA* ioctl command returns non-zero if there is at least one free space left in the receive FIFO.

This command tests the RFF bit of the IIC Status Register.

Returns: UWord16 to be used in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CAN_REQUEST_DATA* command is implemented as a macro.

Example 5-622. IIC_CAN_REQUEST_DATA

```
if(ioctl(IIC, IIC_CAN_REQUEST_DATA, NULL))
{
      ioctl(IIC, IIC_REQUEST_DATA, NULL);
}
```

This code checks if the IIC receiver accepts another byte to its FIFO and, if yes, it requests the data byte to be shifted-in.

5.18.3.20 IIC_READ_REQUEST - master-receive request of a single byte

Call(s):

```
void ioctl(const int *pModuleBase, IIC_READ_REQUEST,
     NULL);
```

Arguments:

Table 5-707. IIC_READ_REQUEST ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_REQUEST* ioctl command requests to clock-in one byte during the master-receive transaction.

This command writes one to the CMD bit of the IIC Data Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_READ_REQUEST* command is implemented as a macro.

Example 5-623. IIC_READ_REQUEST

```
if(ioctl(IIC, IIC_CAN_REQUEST_DATA, NULL))
{
     ioctl(IIC, IIC_READ_REQUEST, NULL);
}
```

This code checks if the IIC receiver accepts another byte to its FIFO and, if yes, it requests the data byte to be shifted-in.

5.18.3.21 IIC_INT_ENABLE - enable interrupts

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INT_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-708. IIC_INT_ENABLE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, use any combination of the following bit values: **error interrupts:** IIC_TXABRT_INT ... transmit abort IIC_TXOVR_INT ... transmit FIFO overrun IIC_RXOVR_INT ... receiver FIFO overrun IIC_RXUND_INT ... receiver FIFO underrun **general call:** IIC_GC_INT ... general call accepted **receiver interrupts:** IIC_RXFULL_INT ... receiver full (FIFO condition configurable) **transmitter interrupts:** IIC_TXDONE_INT ... transmit done IIC_RDREQ_INT ... read request IIC_TXEMPTY_INT ... transmitter empty (FIFO condition configurable) **status interrupts:** IIC_STDET_INT ... start signal detected on bus IIC_STPDET_INT ... stop signal detected on bus IIC_ACT_INT ... bus activity detected

Description: The *IIC_INT_ENABLE* ioctl command enables the selected interrupts to be generated.

This command sets selected bits in the IIC Interrupt Enable Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INT_ENABLE* command is implemented as a macro.

Example 5-624. IIC_INT_ENABLE

```
ioctl(IIC, IIC_INT_ENABLE, IIC_RXFULL_INT | IIC_RDREQ_INT);
```

This code enables the receiver-full and read-request interrupts - the ones needed when operating in slave mode.

5.18.3.22 IIC_INT_DISABLE - disable interrupts

Call(s):

```
void ioctl(const int *pModuleBase, IIC_INT_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-709. IIC_INT_DISABLE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, use any combination of the following bit values: **error interrupts:** IIC_TXABRT_INT ... transmit abort IIC_TXOVR_INT ... transmit FIFO overrun IIC_RXOVR_INT ... receiver FIFO overrun IIC_RXUND_INT ... receiver FIFO underrun **general call:** IIC_GC_INT ... general call accepted **receiver interrupts:** IIC_RXFULL_INT ... receiver full (FIFO condition configurable) **transmitter interrupts:** IIC_TXDONE_INT ... transmit done IIC_RDREQ_INT ... read request IIC_TXEMPTY_INT ... transmitter empty (FIFO condition configurable) **status interrupts:** IIC_STDET_INT ... start signal detected on bus IIC_STPDET_INT ... stop signal detected on bus IIC_ACT_INT ... bus activity detected

Description: The *IIC_INT_DISABLE* ioctl command disables the selected interrupts to be generated.

This command clears selected bits in the IIC Interrupt Enable Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_INT_DISABLE* command is implemented as a macro.

Example 5-625. IIC_INT_DISABLE

```
ioctl(IIC, IIC_INT_DISABLE, IIC_RXFULL_INT | IIC_RDREQ_INT);
```

This code disables the receiver-full and read-request interrupts.

5.18.3.23 IIC_READ_INT_FLAGS - read raw interrupt flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_INT_FLAGS,
      NULL);
```

Arguments:

Table 5-710. IIC_READ_INT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_INT_FLAGS* ioctl command returns the value of the Raw Interrupt Status Register, which collects the interrupt bits. These bits in this register show true status of the IIC module as the bits are NOT masked by disabling the interrupts.

A similar command IIC_TEST_INT_FLAGS may be used to explicitly test a selected bits in the same register.

Returns: Raw Interrupt Status Register value as UWord16.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_READ_INT_FLAGS* command is implemented as a macro.

Example 5-626. IIC_READ_INT_FLAGS

```
if(ioctl(IIC, IIC_READ_INT_FLAGS, NULL) & IIC_STAT_RXOVR)
{
      // IIC receiver overflow happened
}
```

This code polls the receiver overrun error condition.

5.18.3.24 IIC_TEST_INT_FLAGS - test raw interrupt flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_TEST_INT_FLAGS,
      UWord16 param);
```

Arguments:

Table 5-711. IIC_TEST_INT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, use any combination of the following bit values: **error interrupts:** IIC_TXABRT_INT ... transmit abort IIC_TXOVR_INT ... transmit FIFO overrun IIC_RXOVR_INT ... receiver FIFO overrun IIC_RXUND_INT ... receiver FIFO underrun **general call:** IIC_GC_INT ... general call accepted **receiver interrupts:** IIC_RXFULL_INT ... receiver full (FIFO condition configurable) **transmitter interrupts:** IIC_TXDONE_INT ... transmit done IIC_RDREQ_INT ... read request IIC_TXEMPTY_INT ... transmitter empty (FIFO condition configurable) **status interrupts:** IIC_STDET_INT ... start signal detected on bus IIC_STPDET_INT ... stop signal detected on bus IIC_ACT_INT ... bus activity detected

Description: The *IIC_TEST_INT_FLAGS* ioctl command returns non-zero if any of selected interrupts flags is set, regardless if a particular interrupt is enabled or not.

Similarly as the IIC_READ_INT_FLAGS, this command tests the bits of the Raw Interrupt Status Register.

Returns: UWord16 to be interpreted in zero/non-zero conditions only.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_TEST_INT_FLAGS* command is implemented as a macro.

Example 5-627. IIC_TEST_INT_FLAGS

```
if(ioctl(IIC, IIC_TEST_INT_FLAGS, IIC_STAT_RXOVR))
{
      // IIC receiver overflow happened
}
```

This code polls the receiver overrun error condition.

5.18.3.25 IIC_CLEAR_EINT_FLAGS - clear error interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_EINT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-712. IIC_CLEAR_EINT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, any combination of interrupt bits may be specified, but only the following interrupts get actually cleared: IIC_TXABRT_INT ... transmit abort IIC_TXOVR_INT ... transmit FIFO overrun IIC_RXOVR_INT ... receiver FIFO overrun IIC_RXUND_INT ... receiver FIFO underrun

Description: The *IIC_CLEAR_EINT_FLAGS* ioctl command clears the selected error-interrupt flags. This command is typically used in the service routine of the IIC error interrupt to acknowledge the interrupt sources handled.

When the IIC_TXABRT_INT bit is set in the parameter, both the TXABRT interrupt flag and the TXABRTSRC registers are cleared.

Depending on the parameter value, this command performs a dummy read of some or all of the following registers: CLRTXABRT, CLRTXOVR, CLRRXOVR, CLRRXUND.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_EINT_FLAGS* command is implemented as a macro.

Example 5-628. IIC_CLEAR_EINT_FLAGS

```
UWord16 irq = ioctl(IIC, IIC_READ_INT_FLAGS, NULL);

if(irq & IIC_TXABRT_INT)
     // handle tx-abort interrupt
if(irq & IIC_TXOVR_INT)
     // handle tx-overrun interrupt
...

// clear all interrupt flags handled above
ioctl(IIC, IIC_CLEAR_EINT_FLAGS, irq);
```

This code processes and acknowledges the IIC error interrupt sources.

5.18.3.26 IIC_CLEAR_GINT_FLAGS - clear general call interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_GINT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-713. IIC_CLEAR_GINT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, any combination of interrupt bits may be specified, but only the general call interrupts gets actually cleared if the parameter contains the IIC_GC_INT bit.

Description: The *IIC_CLEAR_GINT_FLAGS* ioctl command clears the general-call interrupt flag. This command is typically used in the service routine of the IIC general-call interrupt to acknowledge the interrupt.

Even if there is only one interrupt flag to be cleared with this command, an approach of bit-mask parameter value is used the same way as at the other IIC_CLEAR_xINT_FLAGS commands. This enables multiple IIC interrupt vectors to be routed to a single interrupt service routine and enables the interrupt sources to be handled the same way as described above in the IIC_CLEAR_EINT_FLAGS example.

Depending on if the parameter value contains the IIC_GC_INT bit, this command performs a dummy read of the CLRGC register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_GINT_FLAGS* command is implemented as a macro.

Example 5-629. IIC_CLEAR_GINT_FLAGS

```
// clear GC interrupt
ioctl(IIC, IIC_CLEAR_GINT_FLAGS, IIC_GC_INT);
```

This code acknowledges the IIC general-call interrupt source.

5.18.3.27 IIC_CLEAR_TINT_FLAGS - clear transmitter interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_TINT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-714. IIC_CLEAR_TINT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, any combination of interrupt bits may be specified, but only the following interrupts get actually cleared: IIC_TXDONE_INT ... transmit done IIC_RDREQ_INT ... read request

Description: The *IIC_CLEAR_TINT_FLAGS* ioctl command clears the selected transmitter-interrupt flags. This command is typically used in the service routine of the IIC transmitter interrupt to acknowledge the interrupt sources handled.

The IIC_TXEMPTY_INT interrupt source can NOT be cleared with this command. This interrupt flag is cleared automatically when TXFIFO is filled with a new data. If no more bytes are to be transmitted, the IIC_TXEMPTY_INT interrupt should be disabled using the IIC_INT_DISABLE command. The similar approach is applied to the IIC_RXFULL_INT interrupt (there is no way to explicitly clear this interrupt, the RX FIFO needs to be read to clear this flag).

Depending on the parameter value, this command performs a dummy read of some or all of the following registers: CLRTXDONE, CLRRDREQ.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_TINT_FLAGS* command is implemented as a macro.

Example 5-630. IIC_CLEAR_TINT_FLAGS

```
UWord16 irq = ioctl(IIC, IIC_READ_INT_FLAGS, NULL);

if(irq & IIC_TXDONET_INT)
     // handle tx-done interrupt
if(irq & IIC_RDREQ_INT)
     // handle read request interrupt
...

// clear all interrupt flags handled above
ioctl(IIC, IIC_CLEAR_TINT_FLAGS, irq);
```

This code processes and acknowledges the IIC transmitter interrupt sources.

5.18.3.28 IIC_CLEAR_SINT_FLAGS - clear status interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_SINT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-715. IIC_CLEAR_SINT_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	UWord16, any combination of interrupt bits may be specified, but only the following interrupts get actually cleared: IIC_STDET_INT ... start signal detected on bus IIC_STPDET_INT ... stop signal detected on bus IIC_ACT_INT ... bus activity detected

Description: The *IIC_CLEAR_SINT_FLAGS* ioctl command clears the selected status-interrupt flags. This command is typically used in the service routine of the IIC status interrupt to acknowledge the interrupt sources handled.

Depending on the parameter value, this command performs a dummy read of some or all of the following registers: CLRSTDET, CLRSTPDET, CLRACT.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_SINT_FLAGS* command is implemented as a macro.

Example 5-631. IIC_CLEAR_SINT_FLAGS

```
UWord16 irq = ioctl(IIC, IIC_READ_INT_FLAGS, NULL);

if(irq & IIC_STDET_INT)
     // handle start-detected interrupt
if(irq & IIC_STPDET_INT)
     // handle stop-detected interrupt
...

// clear all interrupt flags handled above
ioctl(IIC, IIC_CLEAR_SINT_FLAGS, irq);
```

This code processes and acknowledges the IIC status interrupt sources.

5.18.3.29 IIC_CLEAR_ALL_INTS - clear all interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_ALL_INTS, NULL);
```

Arguments:

Table 5-716. IIC_CLEAR_ALL_INTS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CLEAR_ALL_INTS* ioctl command clears all interrupt flags and also the TXABRTSRC Register.

This command performs a dummy read of the CLRINT Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_ALL_INTS* command is implemented as a macro.

Example 5-632. IIC_CLEAR_ALL_INTS

```
// clear all interrupt flags
ioctl(IIC, IIC_CLEAR_ALL_INTS, NULL);
```

This code clears all IIC interrupt sources and clears the TXABRTSRC Register.

5.18.3.30 IIC_READ_TXABORT_SOURCE - read transmit abort source register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_TXABORT_SOURCE,
     NULL);
```

Arguments:

Table 5-717. IIC_READ_TXABORT_SOURCE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_TXABORT_SOURCE* ioctl command reads and returns the value of the IIC Transmit Abort Source Register. The returned value may be tested using any of the IIC_TXABRTSRC_*xxx* bits.

Returns: UWord16 as a content of the TXABRTSRC Register.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_READ_TXABORT_SOURCE* command is implemented as a macro.

Example 5-633. IIC_READ_TXABORT_SOURCE

```
UWord16 abrtsrc = ioctl(IIC, IIC_READ_TXABORT_SOURCE, NULL);
```

This code reads the Transmitter Abort Source Register.

5.18.3.31 IIC_CLEAR_TXABORT_SOURCE - clear transmit abort source register and interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, IIC_CLEAR_TXABORT_SOURCE,
     NULL);
```

Arguments:

Table 5-718. IIC_CLEAR_TXABORT_SOURCE ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_CLEAR_TXABORT_SOURCE* ioctl command clears the TXABRT interrupt flag and the TXABRTSRC Register.

This command performs a dummy read of the CLRTXABRT Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_CLEAR_TXABORT_SOURCE* command is implemented as a macro.

Example 5-634. IIC_CLEAR_TXABORT_SOURCE

```
ioctl(IIC, IIC_CLEAR_TXABORT_SOURCE, NULL);
```

This code clears the Transmitter Abort Source register and interrupt flag.

5.18.3.32 IIC_READ_STATUS_FLAGS - read status register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_READ_STATUS_FLAGS,
     NULL);
```

Arguments:

Table 5-719. IIC_READ_STATUS_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.

Description: The *IIC_READ_STATUS_FLAGS* ioctl command reads and returns the value of the IIC Status Register. The returned value may be tested using any of the IIC_STAT_*xxx* bits.

See the IIC_TEST_STATUS_FLAGS for the list of the IIC_STAT_*xxx* values.

Returns: UWord16 as a content of the STAT Register.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_READ_STATUS_FLAGS* command is implemented as a macro.

Example 5-635. IIC_READ_STATUS_FLAGS

```
if(ioctl(IIC, IIC_READ_STATUS_FLAGS, NULL) & IIC_STAT_ACT)
{
     // IIC is active
}
```

This code reads the IIC Status Register and tests the “Activity” (ACT) bit.

5.18.3.33 IIC_TEST_STATUS_FLAGS - test status register bits

Call(s):

```
UWord16 ioctl(const int *pModuleBase, IIC_TEST_STATUS_FLAGS,
      UWord16 param);
```

Arguments:

Table 5-720. IIC_TEST_STATUS_FLAGS ioctl call arguments

*pModuleBase	in	The IIC module identifier. Use IIC.
param	in	Any combination of the following values: IIC_STAT_SLVACT ... slave activity status IIC_STAT_MSTACT ... master activity status IIC_STAT_RFF ... receive FIFO completely full IIC_STAT_RFNE ... receive FIFO not empty IIC_STAT_TFE ... transmit FIFO completely empty IIC_STAT_TFNF ... transmit FIFO not full IIC_STAT_ACT ... slave or master activity

Description: The *IIC_TEST_STATUS_FLAGS* ioctl command returns a non-zero value if any of the bits selected by the parameter are set in the IIC Status Register.

Returns: UWord16 value to be tested in zero/non-zero conditions.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *IIC_TEST_STATUS_FLAGS* command is implemented as a macro.

Example 5-636. IIC_TEST_STATUS_FLAGS

```
if(ioctl(IIC, IIC_TEST_STATUS_FLAGS, IIC_STAT_ACT))
{
      // IIC is active
}
```

This code tests “Activity” (ACT) bit in the IIC Status Register.

5.18.4 IIC Driver Applications

There is one sample application demonstrating how the IIC module can be used in a user application. The application is compatible with the sample application for the MC56F801x devices. Please see the description of this application in section Section 5.17.4. The supported EVM boards for the IIC sample application are MC56F8025DEMO and MC56F8037EVM.

Please see the application source code in
{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\iic_demo

5.19 Temperature Sensor System Driver

This section describes the API for the MC56F83xx Temperature Sensor System on-chip module. The functionality of the Temperature Sensor System module itself is described in the **MC56F8300 Peripheral User Manual**.

5.19.1 Introduction

The Temperature Sensor module intended to operate in conjunction with an on-chip Analog-to-Digital Converter (ADC). The module is composed of a custom analog block and a simple IPBus interface. Temperature readings may be taken through the ADC. The ADC includes the ability to continuously monitor an input and issue an interrupt when a limit is exceeded. This provides the ability to easily implement an over-temperature alert.

This section describes the Temperature Sensor System driver software providing the lowest level software layer, interfacing hardware to software.

5.19.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.19.3.

Table 5-721. Temperature Sensor System Module Base Address

Module base address of / for	MC56F8013	MC56F8322 MC56F8323	MC56F8345 MC56F8346 MC56F8347	MC56F8356 MC56F8357	MC56F8367
TSENSOR (TSNSR_BASE)	N/A	0xF270	0xF270	0xF270	0xF270

5.19.2.1 API Definition

The following header files are needed in order to use the Temperature Sensor device driver:

Required Header File(s):

```
#include "qs.h"
#include "tsensor.h"
```

The following information may be found in the header file *tsensor.h*.

Public Data Structure(s):

None

5.19.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static Temperature Sensor module configuration by the driver initialization routine. These symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-722. Temperature Sensor System Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
TSENSOR_CNTL_INIT	UWord16	Represents contents of the Temperature Sensor Control Register

5.19.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void Cmd, void param);
void ioctl(const int *pModuleBase, void Cmd, void param);
```

Description: The *ioctl* call "modifies" the Temperature Sensor device modes or accesses the Temperature Sensor System register(s).

Arguments:

Table 5-723. Temperature Sensor Driver Arguments - ioctl

pModuleBase	in	Temperature Sensor module identifier. Use TSENSOR.
Cmd	in	Commands found in *tsensor.h* which are used to modify the Temperature Sensor module status and control registers. See Table 5-724.

Table 5-723. Temperature Sensor Driver Arguments - ioctl

pParam, param	in, out, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-724. ioctl commands

Cmd	pParam, param	Return	Description
TSENSOR_INIT	NULL	None	Initializes Temperature Sensor System module by data from configuration file (appconfig.h).
TSENSOR_DEVICE	TSENSOR_POWER_ON / TSENSOR_POWER_OFF	None	Powers on or off the current source I_c of the temperature sensor.
TSENSOR_IS_POWERED_ON	NULL	UWord16	Returns the status of the Temperature Sensor System.

5.19.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the ioctl commands usage.

5.19.3.1 TSENSOR_INIT - initialize Temperature Sensor System module

Call(s):

```
void ioctl(const int *pModuleBase, TSENSOR_INIT, NULL);
```

Arguments:

Table 5-725. TSENSOR_INIT ioctl call arguments

*pModuleBase	in	The Temperature Sensor System module identifier. Use TSENSOR.

Description: The *TSENSOR_INIT* ioctl command is used to initialize the Temperature Sensor System module. Initialization consists of writing the initialization value to the Temperature Sensor Control Register (TSENSOR_CONTROL). Initialization value (configuration item) is taken from the *appconfig.h* file. If no initialization value is defined in *appconfig.h,* then no initialization is performed. For reference on symbol of configuration item see Section 5.19.2.2.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *TSENSOR_INIT* ioctl command is implemented as a function.

Example 5-637. TSENSOR_INIT

```
ioctl(TSENSOR, TSENSOR_INIT, NULL);
```

This code initializes the Temperature Sensor System module with the initialization value from *appconfig.h*.

5.19.3.2 TSENSOR_DEVICE - power on or off the temperature sensor

Call(s):

```
void ioctl(const int *pModuleBase, TSENSOR_DEVICE, UWord16 param);
```

Arguments:

Table 5-726. TSENSOR_DEVICE ioctl call arguments

*pModuleBase	in	The Temperature Sensor System module identifier. Use TSENSOR.
param	in	Parameter to select the desired action. Use TSENSOR_POWER_ON to power on the temperature sensor or TSENSOR_POWER_OFF to power off it.

Description: The *TSENSOR_DEVICE* ioctl command powers on or powers off the current source I_c of the temperature sensor. This command sets the PWR bit (Bit 0) in the Temperature Sensor Control Register when TSENSOR_POWER_ON is used as a parameter. This command clears the above mentioned bit in case of TSENSOR_POWER_OFF.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *TSENSOR_DEVICE* ioctl command is implemented as a macro.

Example 5-638. TSENSOR_DEVICE

```
ioctl(TSENSOR, TSENSOR_DEVICE, TSENSOR_POWER_ON);
```

This code powers on the Temperature Sensor device.

5.19.3.3 TSENSOR_IS_POWERED_ON - return the status of the Temperature Sensor System

Call(s):

```
UWord16 ioctl(const int *pModuleBase, TSENSOR_IS_POWERED_ON,
          NULL);
```

Arguments:

Table 5-727. TSENSOR_IS_POWERED_ON ioctl call arguments

*pModuleBase	in	The Temperature Sensor System module identifier. Use TSENSOR.

Description: The *TSENSOR_IS_POWERED_ON* ioctl command returns the status of the Temperature Sensor System - logic value “1” when temperature sensor is powered, or logic value “0” when temperature sensor is not powered.

Returns: status of the Temperature Sensor as UWord16.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *TSENSOR_IS_POWERED_ON* ioctl command is implemented as a macro.

Example 5-639. TSENSOR_IS_POWERED_ON

```
UWord16    tSensorPwr;

tSensorPwr = ioctl(TSENSOR, TSENSOR_IS_POWERED_ON, NULL);
```

This code stores the status of the Temperature Sensor System to variable tSensorPwr.

5.19.4 Temperature Sensor System Driver Application

N/A

5.20 Quad Timer Driver

This section describes the API for the MC56F83xx and the MC56F80xx Quad Timer on-chip module. The functionality of the Quad Timer module itself is described in the **MC56F83xx Peripheral User Manual**, **56F8000 Peripheral Reference Manual** and **56F802x/3x Peripheral Reference Manual**.

5.20.1 Introduction

The MC56F8xxx can have up to four Quad Timer modules: A, B, C and D. Note that in case of a single Quad Timer module on the silicon, it can be labeled without alphabetic character. Each Quad Timer module consists of four identical 16-bit timers/counters with capture/compare capabilities. This section describes the Quad Timer driver software providing the lowest level software layer, interfacing the hardware with the software.

5.20.2 Quick Reference

This section is intended to be a source of quick access information while the details are discussed in Section 5.20.3.

Table 5-728. Quad Timer Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F8037	MC56F832x	MC56F83xx
QTIMER_A (TMRA_BASE)	N/A	N/A	0xF000	0xF000	0xF040	0xF040
QTIMER_B (TMRB_BASE)	N/A	N/A	N/A	0xF040	N/A	0xF080
QTIMER_C (TMRC_BASE)	N/A	N/A	N/A	N/A	0xF0C0	0xF0C0
QTIMER_D (TMRD_BASE)	N/A	N/A	N/A	N/A	N/A	0xF100
QTIMER (TMR_BASE)	0xF000	0xF000	N/A	N/A	N/A	N/A

5.20.2.1 API Definition

The following header files are needed in order to use the Quad Timer device driver:

Required Header File(s):

```
#include "qs.h"
#include "qtimer.h"
```

The following information may be found in the header file *qtimer.h*.

Public Data Structure(s):

none

5.20.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static configuration of the Quad Timer module by the driver initialization routine. These symbols are intended for the application (project) specific configuration file *appconfig.h*. See e.g. Example 5-688 for more details.

Table 5-729. Quad Timer Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
QT_xx_CTRL_INIT	UWord16	Initial value of the timer/counter Control Register (TMRCTRL)
QT_xx_SCR_INIT	UWord16	Initial value of the timer/counter Status and Control Register (TMRSCR)
QT_xx_CMP1_INIT	UWord16	Initial value of the timer/counter Compare Register 1 (TMRCMP1)
QT_xx_CMP2_INIT	UWord16	Initial value of the timer/counter Compare Register 2 (TMRCMP2)
QT_xx_LOAD_INIT	UWord16	Initial value of the timer/counter Load Register (TMRLOAD)
QT_xx_CNTR_INIT	UWord16	Initial value of the timer/counter Counter Register (TMRCNTR)
QT_xx_CMPLD1_INIT	UWord16	Initial value of the Timer Comparator Load Registers 1 (TMRCMPLD1)
QT_xx_CMPLD2_INIT	UWord16	Initial value of the Timer Comparator Load Registers 2 (TMRCMPLD2)
QT_xx_COMSCR_INIT	UWord16	Initial value of the Timer Comparator Status/Control Registers (TMRCOMSCR)
MC56F802x/3x only: QT_x0_ENBL_INIT *(channel 0 only)*	UWord16	Initial value of the Channel Enable Register
MC56F802x/3x only: QT_xx_FILT_INIT	UWord16	Initial value of the Input Filter Register

Note - "xx" specifies the used timer. It can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or just 0, 1, 2, 3 in case of single timer.

5.20.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, void *pParam);
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
```

Description: The *ioctl* call "changes" the timer/counter modes or accesses the timer/counter register(s).

Arguments:

Table 5-730. Quad Timer Driver Arguments - ioctl

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 in case of single timer.
cmd	in	Commands found in *qtimer.h* which are used to modify the timer/counter registers. See Table 5-731.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separator Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-731. ioctl commands

cmd	param / pParam	Return	Description
QT_INIT	NULL	None	Initializes timer/counter by data from configuration file (appcon-fig.h).

Table 5-731. ioctl commands (Continued)

cmd	param / pParam	Return	Description
QT_SET_COUNT_MODE	use one of the following values: **standard modes:** QT_NO_OPERATION / QT_COUNT_RISING_EDGES_MODE / QT_COUNT_BOTH_EDGES_MODE / QT_GATED_COUNT_MODE / QT_QUADRATURE_COUNT_MODE / QT_SIGNED_COUNT_MODE / QT_TRIGGERED_COUNT_MODE / QT_CASCADE_COUNT_MODE / **special modes:** QT_ONE_SHOT_MODE / QT_PULSE_OUTPUT_MODE / QT_FIXED_FREQ_PWM_MODE / QT_VARIABLE_FREQ_PWM_MODE	None	Sets timer/counter count mode.
QT_SET_PRIMARY_SOURCE	QT_COUNTER0_INPUT / QT_COUNTER1_INPUT / QT_COUNTER2_INPUT / QT_COUNTER3_INPUT / QT_COUNTER0_OUTPUT/ QT_COUNTER1_OUTPUT/ QT_COUNTER2_OUTPUT/ QT_COUNTER3_OUTPUT/ QT_IPBUS_DIV_1 / QT_IPBUS_DIV_2 / QT_IPBUS_DIV_4 / QT_IPBUS_DIV_8 / QT_IPBUS_DIV_16 / QT_IPBUS_DIV_32 / QT_IPBUS_DIV_64 / QT_IPBUS_DIV_128	None	Sets primary count source of timer/counter.
QT_SET_SECONDARY_SOURCE	QT_COUNTER0_INPUT / QT_COUNTER1_INPUT / QT_COUNTER2_INPUT / QT_COUNTER3_INPUT	None	Sets secondary count source of timer/counter.
QT_SET_COUNT_ONCE	QT_COUNT_REPEATEDLY / QT_UNTIL_COMPARE_AND_STOP	None	Sets continuous or one shot counting mode.
QT_SET_COUNT_LENGTH	QT_ROLL_OVER / QT_UNTIL_COMPARE_AND_REINIT	None	Sets count length of timer/counter.
QT_SET_COUNT_DIRECTION	QT_COUNT_UP / QT_COUNT_DOWN	None	Sets whether timer/counter is counting up or down.
QT_CO_CHANNEL_INIT	QT_ENABLE / QT_DISABLE	None	Enables/disables another timer in the same Quad Timer module to force re-initialization of this timer when it has encountered an active compare event.

Table 5-731. ioctl commands (Continued)

cmd	param / pParam	Return	Description
QT_SET_OUTPUT_MODE	QT_SET_WHILE_ACTIVE / QT_CLEAR_ON_COMPARE / QT_SET_ON_COMPARE / QT_TOGGLE_ON_COMPARE / QT_TOGGLE_USING_ALT_COMPARE / QT_CLEAR_ON_SECONDARY / QT_CLEAR_ON_COUNTER_ROLLOVER / QT_GATED_CLOCK_WHILE_ACTIVE	None	Sets timer/counter output mode (i.e. mode of operation for the OFLAG output signal).
QT_CLEAR_FLAG	QT_COMPARE_FLAG | QT_OVERFLOW_FLAG | QT_INPUT_EDGE_FLAG | QT_VAL_FLAG	None	Clears selected timer/counter flags.
QT_CLEAR_COMPARE_FLAG	QT_COMPARE1_FLAG | QT_COMPARE2_FLAG	None	Clears selected timer/counter compare flags.
QT_READ_FLAG	QT_COMPARE_FLAG | QT_OVERFLOW_FLAG | QT_INPUT_EDGE_FLAG | QT_VAL_FLAG	UWord16	Returns the status of selected timer/counter flags.
QT_READ_COMPARE_FLAG	QT_COMPARE1_FLAG | QT_COMPARE2_FLAG	UWord16	Returns the status of selected timer/counter compare flags.
QT_INT_ENABLE	QT_COMPARE_INT | QT_COMPARE1_INT | QT_COMPARE2_INT | QT_OVERFLOW_INT | QT_INPUT_EDGE_INT	None	Enables selected interrupts.
QT_INT_DISABLE	QT_COMPARE_INT | QT_COMPARE1_INT | QT_COMPARE2_INT | QT_OVERFLOW_INT | QT_INPUT_EDGE_INT	None	Disables selected interrupts.
QT_SET_INPUT_POLARITY	QT_NORMAL_POLARITY / QT_INVERTED_POLARITY	None	Sets normal or inverted input signal polarity.
QT_READ_EXT_INPUT_PIN	NULL	0x0000 - input pin is low 0x0100 - input pin is high	Reads the current state of the external input pin after application of the Input Polarity Select (IPS) bit.
QT_SET_CAPTURE_MODE	QT_CAPTURE_DISABLED / QT_RISING_EDGE / QT_FALLING_EDGE / QT_BOTH_EDGES	None	Sets Input Capture Mode.
QT_MASTER_MODE	QT_ENABLE / QT_DISABLE	None	Enables/disables Master Mode.
QT_EXT_OFLAG_FORCE	QT_ENABLE / QT_DISABLE	None	Enables/disables external OFLAG force.

Table 5-731. ioctl commands (Continued)

cmd	param / pParam	Return	Description
QT_FORCE_OFLAG	QT_ZERO / QT_ONE	None	Clears/sets OFLAG output. This command should be used only if timer/counter is disabled.
QT_SET_OUTPUT_POLARITY	QT_NORMAL_POLARITY / QT_INVERTED_POLARITY	None	Sets the polarity of the OFLAG output signal.
QT_OUTPUT_ON_EXT_PIN	QT_ENABLE / QT_DISABLE	None	Enables/disables the OFLAG output to be put on the external pin.
QT_SET_LOAD_CONTROL1	QT_NEVER_PRELOAD / QT_LOAD_ON_CMP1 / QT_LOAD_ON_CMP2	None	Specifies the preload event for the Compare Registers 1.
QT_SET_LOAD_CONTROL2	QT_NEVER_PRELOAD / QT_LOAD_ON_CMP1 / QT_LOAD_ON_CMP2	None	Specifies the preload event for the Compare Registers 2.
QT_WRITE_CONTROL_REG	UWord16	None	Writes 16-bit value to the timer/counter Control Register (TMRCTRL).
QT_WRITE_STATUS_CONTROL_REG	UWord16	None	Writes 16-bit value to the timer/counter Status and Control Register (TMRSCR).
QT_WRITE_CMP_STATUS_CONTROL_REG	UWord16	None	Writes 16-bit value to the Timer Comparator Status/Control Registers (TMRCOMSCR).
QT_WRITE_COMPARE_REG1	UWord16	None	Writes 16-bit value to the timer/counter Compare Register 1 (TMRCMP1).
QT_WRITE_COMPARE_REG2	UWord16	None	Writes 16-bit value to the timer/counter Compare Register 2 (TMRCMP2).
QT_WRITE_PRELOAD_COMPARE_REG1	UWord16	None	Writes 16-bit value to the Timer Comparator Load Registers 1 (TMRCMPLD1).
QT_WRITE_PRELOAD_COMPARE_REG2	UWord16	None	Writes 16-bit value to the Timer Comparator Load Registers 2 (TMRCMPLD2).
QT_WRITE_LOAD_REG	UWord16	None	Writes 16-bit value to the timer/counter Load Register (TMRLOAD).
QT_WRITE_COUNTER_REG	UWord16	None	Writes 16-bit value to the timer/counter Counter Register (TMRCNTR).

Table 5-731. ioctl commands (Continued)

cmd	param / pParam	Return	Description
QT_READ_CONTROL_REG	NULL	UWord16	Returns the content of the timer/counter Control Register (TMRCTRL).
QT_READ_STATUS_CONTROL_ REG	NULL	UWord16	Returns the content of the timer/counter Status and Control Register (TMRSCR).
QT_READ_CMP_STATUS_ CONTROL_REG	NULL	UWord16	Returns the content of the Timer Comparator Status/Control Registers (TMRCOMSCR).
QT_READ_COMPARE_REG1	NULL	UWord16	Returns the content of the timer/counter Compare Register 1 (TMRCMP1).
QT_READ_COMPARE_REG2	NULL	UWord16	Returns the content of the timer/counter Compare Register 2 (TMRCMP2).
QT_READ_LOAD_REG	NULL	UWord16	Returns the content of the timer/counter Load Register (TMRLOAD).
QT_READ_COUNTER_REG	NULL	UWord16	Returns the content of the timer/counter Counter Register (TMRCNTR).
QT_READ_CAPTURE_REG	NULL	UWord16	Returns the content of the timer/counter Capture Register (TMRCAP).
QT_READ_HOLD_REG	NULL	UWord16	Returns the content of the timer/counter Hold Register (TMRHOLD).
only on MC56F802x/3x: QT0_MASS_ENABLE *(only applicable to timer channel 0)*	QT_CH0 | QT_CH1 | QT_CH2 | QT_CH3	None	Simultaneously enables the selected timer channels.
only on MC56F802x/3x: QT0_MASS_DISABLE *(only applicable to timer channel 0)*	QT_CH0 | QT_CH1 | QT_CH2 | QT_CH3	None	Simultaneously disables the selected timer channels.
only on MC56F802x/3x: QT_READ_FILT_REG	NULL	UWord16	Returns value of the Input Filter Register.
only on MC56F802x/3x: QT_WRITE_FILT_REG	UWord16	None	Writes given value to the Input Filter Register.

5.20.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code example illustrates the usage of the ioctl commands.

5.20.3.1 QT_INIT - initialize timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_INIT, NULL);
```

Arguments:

Table 5-732. QT_INIT ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_INIT* ioctl command is used to initialize the selected timer/counter. The initialization consists of writing to these timer/counter registers: Control Register (TMRCTRL), Status and Control Register (TMRSCR), Compare Register 1 (TMRCMP1), Compare Register 2 (TMRCMP2), Load Register (TMRLOAD) and Counter Register (TMRCNTR). The initialization values (configuration items) for each register are taken from *appconfig.h*, each configuration item corresponds to one register. If no initialization value (configuration item) is defined in *appconfig.h*, then no initialization is done. It is not necessary to define initialization values for all timer/counter registers in *appconfig.h*, define only those which are needed. For reference on the names of configuration items see Section 5.20.2.2, “Configuration Items.”

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_INIT* ioctl command is implemented as a function call.

Example 5-640. QT_INIT

```
ioctl(QTIMER_D1, QT_INIT, NULL);
```

This code initializes timer/counter D1 with the initialization values from *appconfig.h*.

5.20.3.2 QT_SET_COUNT_MODE - set timer/counter count mode

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_COUNT_MODE,
           UWord16 Param);
```

Arguments:

Table 5-733. QT_SET_COUNT_MODE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	The timer/counter mode of operation. Use one of these predefined constants: **standard modes:** QT_NO_OPERATION / QT_COUNT_RISING_EDGES_MODE / QT_COUNT_BOTH_EDGES_MODE / QT_GATED_COUNT_MODE / QT_QUADRATURE_COUNT_MODE / QT_SIGNED_COUNT_MODE / QT_TRIGGERED_COUNT_MODE / QT_CASCADE_COUNT_MODE / **special modes:** QT_ONE_SHOT_MODE / QT_PULSE_OUTPUT_MODE / QT_FIXED_FREQ_PWM_MODE / QT_VARIABLE_FREQ_PWM_MODE

Description: The *QT_SET_COUNT_MODE* ioctl command sets the timer/counter count mode by writing the mode of operation (selected by parameter Param) to the timer/counter Control Register (TMRCTRL).

All modes affect the TMRCTRL bits 15-13 (Count Mode).

Special modes affect also bits 2-0 (Output Mode), bit 6 (ONCE) and bit 5 (LENGTH). Bit 5 is not affected by *QT_PULSE_OUTPUT_MODE*.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_COUNT_MODE* ioctl command is implemented as a macro.

Example 5-641. QT_SET_COUNT_MODE

```
ioctl(QTIMER_D0, QT_SET_COUNT_MODE, QT_COUNT_RISING_EDGES_MODE);
```

This code sets the timer/counter D0 count mode to “Count rising edges of primary source“. The timer/counter starts counting immediately.

```
ioctl(QTIMER_D0, QT_SET_COUNT_MODE, QT_NO_OPERATION);
```

This code stops the timer/counter D0.

5.20.3.3 QT_SET_PRIMARY_SOURCE - set primary count source of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_PRIMARY_SOURCE,
           UWord16 Param);
```

Arguments:

Table 5-734. QT_SET_PRIMARY_SOURCE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	The timer/counter primary count source. Use one of these predefined constants: QT_COUNTER0_INPUT / QT_COUNTER1_INPUT / QT_COUNTER2_INPUT / QT_COUNTER3_INPUT / QT_COUNTER0_OUTPUT / QT_COUNTER1_OUTPUT / QT_COUNTER2_OUTPUT / QT_COUNTER3_OUTPUT / QT_IPBUS_DIV_1 / QT_IPBUS_DIV_2 / QT_IPBUS_DIV_4 / QT_IPBUS_DIV_8 / QT_IPBUS_DIV_16 / QT_IPBUS_DIV_32 / QT_IPBUS_DIV_64 / QT_IPBUS_DIV_128

Description: The *QT_SET_PRIMARY_SOURCE* ioctl command is used to set the primary count source of the selected timer/counter. The command writes to the timer/counter Control Register (TMRCTRL), Bits 12-9 (Primary Count Source).

Returns: None.

Range Issues: None.

Special Issues: If it is used as a primary count source input or output, then, for example, the predefined constant QT_COUNTER0_INPUT means input of timer/counter 0 from the same Quad Timer module as the timer/counter is selected by parameter pModuleBase. It is not possible to select the primary count source input or output of the timer/counter from another Quad Timer module.

Design/Implementation: The *QT_SET_PRIMARY_SOURCE* ioctl command is implemented as a macro.

Example 5-642. QT_SET_PRIMARY_SOURCE

```
ioctl(QTIMER_B3, QT_SET_PRIMARY_SOURCE, QT_COUNTER0_OUTPUT);
```

This code selects the primary count source timer/counter B0 output for timer/counter B3.

5.20.3.4 QT_SET_SECONDARY_SOURCE - set secondary count source of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_SECONDARY_SOURCE,
           UWord16 Param);
```

Arguments:

Table 5-735. QT_SET_SECONDARY_SOURCE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	The timer/counter secondary count source. Use one of these pre-defined constants: QT_COUNTER0_INPUT / QT_COUNTER1_INPUT / QT_COUNTER2_INPUT / QT_COUNTER3_INPUT

Description: The *QT_SET_SECONDARY_SOURCE* ioctl command is used to set the secondary count source of the selected timer/counter. The command writes to the timer/counter Control Register (TMRCTRL), Bits 8-7 (Secondary Count Source). The secondary Count Source provides additional information, used for counting. Its meaning depends on the selected timer/counter mode of operation (see **MC56F8300 Peripheral User Manual** or **56F8000 Peripheral Reference Manual**).

Returns: None.

Range Issues: None.

Special Issues: The name COUNTER0 (COUNTER1 etc.) in the predefined constants, used for the Param parameter, always means timer/counter 0 (timer/counter 1 etc.) from the same Quad Timer module as the timer/counter, which is selected by the parameter pModuleBase.

Design/Implementation: The *QT_SET_SECONDARY_SOURCE* ioctl command is implemented as a macro.

Example 5-643. QT_SET_SECONDARY_SOURCE

```
ioctl(QTIMER_B3, QT_SET_SECONDARY_SOURCE, QT_COUNTER0_INPUT);
```

This code selects the secondary count source timer/counter B0 input pin for timer/counter B3.

5.20.3.5 QT_SET_COUNT_ONCE - set continuous or one shot counting mode of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_COUNT_ONCE,
           UWord16 Param);
```

Arguments:

Table 5-736. QT_SET_COUNT_ONCE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	The timer/counter count once mode. Use one of these predefined constants: QT_COUNT_REPEATEDLY / QT_UNTIL_COMPARE_AND_STOP

Description: The *QT_SET_COUNT_ONCE* ioctl command is used to set whether the selected timer/counter counts repeatedly or counts until compare successful and then stops. This command clears the ONCE bit (Bit 6) in the timer/counter Control Register (TMRCTRL) when QT_COUNT_REPEATEDLY is used as a parameter. This command sets the above mentioned bit if QT_UNTIL_COMPARE_AND_STOP is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_COUNT_ONCE* ioctl command is implemented as a macro.

Example 5-644. QT_SET_COUNT_ONCE

```
ioctl(QTIMER_C0, QT_SET_COUNT_ONCE, QT_COUNT_REPEATEDLY);
```

This code selects the continuous count mode (count repeatedly) for timer/counter C0.

5.20.3.6 QT_SET_COUNT_LENGTH - set count length of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_COUNT_LENGTH,
           UWord16 Param);
```

Arguments:

Table 5-737. QT_SET_COUNT_LENGTH ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select count length mode. Use one of these predefined constants: QT_ROLL_OVER / QT_UNTIL_COMPARE_AND_REINIT

Description: The *QT_SET_COUNT_LENGTH* ioctl command is used to set whether the selected timer/counter counts up to the compare value and then re-initializes itself to the value specified in the load register (parameter QT_UNTIL_COMPARE_AND_REINIT used), or the counter continues counting past the compare value, up to the binary roll over (parameter QT_ROLL_OVER used). This command clears the LENGTH bit (Bit 5) in the timer/counter Control Register (TMRCTRL) when QT_ROLL_OVER is used as a parameter. This command sets the above mentioned bit if QT_UNTIL_COMPARE_AND_REINIT is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_COUNT_LENGTH* ioctl command is implemented as a macro.

Example 5-645. QT_SET_COUNT_LENGTH

```
ioctl(QTIMER_C0, QT_SET_COUNT_LENGTH, QT_ROLL_OVER);
```

This code sets the binary roll over mode for timer/counter C0.

5.20.3.7 QT_SET_COUNT_DIRECTION - set count direction of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_COUNT_DIRECTION,
           UWord16 Param);
```

Arguments:

Table 5-738. QT_SET_COUNT_DIRECTION ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the count direction. Use one of these predefined constants: QT_COUNT_UP / QT_COUNT_DOWN

Description: The *QT_SET_COUNT_DIRECTION* ioctl command is used to set whether the selected timer/counter counts up or down. This command clears the DIR bit (Bit 4) in the timer/counter Control Register (TMRCTRL) when QT_COUNT_UP is used as a parameter. This command sets the above mentioned bit if QT_COUNT_DOWN is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_COUNT_DIRECTION* ioctl command is implemented as a macro.

Example 5-646. QT_SET_COUNT_DIRECTION

```
ioctl(QTIMER_A0, QT_SET_COUNT_DIRECTION, QT_COUNT_DOWN);
```

This code sets the counting down mode for timer/counter A0.

5.20.3.8 QT_CO_CHANNEL_INIT - enable/disable co-channel initialization of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_CO_CHANNEL_INIT,
           UWord16 Param);
```

Arguments:

Table 5-739. QT_CO_CHANNEL_INIT ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the desired action. Use one of these predefined constants: QT_ENABLE / QT_DISABLE

Description: The *QT_CO_CHANNEL_INIT* ioctl command enables/disables another timer/counter in the same Quad Timer module to force the re-initialization of this timer/counter, when it has encountered an active compare event (co-channel initialization). This command clears the CoInit bit (Bit 3) in the timer/counter Control Register (TMRCTRL) when QT_DISABLE is used as a parameter. This command sets the above mentioned bit if QT_ENABLE is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_CO_CHANNEL_INIT* ioctl command is implemented as a macro.

Example 5-647. QT_CO_CHANNEL_INIT

```
ioctl(QTIMER_A0, QT_CO_CHANNEL_INIT, QT_ENABLE);
```

This code enables co-channel initialization of timer/counter A0. If any of the A1, A2 or A3 encounters an active compare event and have its Master Mode enabled (i.e. its compare event is broadcasted to other timers/counters in the same Quad Timer module).

5.20.3.9 QT_SET_OUTPUT_MODE - set output mode of timer/counter

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_OUTPUT_MODE,
           UWord16 Param);
```

Arguments:

Table 5-740. QT_SET_OUTPUT_MODE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the output mode. Use one of these predefined constants: QT_SET_WHILE_ACTIVE / QT_CLEAR_ON_COMPARE / QT_SET_ON_COMPARE / QT_TOGGLE_ON_COMPARE / QT_TOGGLE_USING_ALT_COMPARE / QT_CLEAR_ON_SECONDARY / QT_CLEAR_ON_COUNTER_ROLLOVER / QT_GATED_CLOCK_WHILE_ACTIVE

Description: The *QT_SET_OUTPUT_MODE* ioctl command is used to set the output mode of the selected timer/counter (i.e. mode of operation for the OFLAG output signal). The command writes to the timer/counter Control Register (TMRCTRL), Bits 2-0 (Output Mode).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_OUTPUT_MODE* ioctl command is implemented as a macro.

Example 5-648. QT_SET_OUTPUT_MODE

```
ioctl(QTIMER_A0, QT_SET_OUTPUT_MODE, QT_TOGGLE_ON_COMPARE);
```

This code selects the output mode of the timer/counter A0 to toggle on compare.

5.20.3.10 QT_CLEAR_FLAG - clear timer/counter flags

Call(s):

```
void ioctl(const int *pModuleBase, QT_CLEAR_FLAG, UWord16 Param);
```

Arguments:

Table 5-741. QT_CLEAR_FLAG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter flag. Use consolidation of these predefined constants: QT_COMPARE_FLAG | QT_OVERFLOW_FLAG | QT_INPUT_EDGE_FLAG | QT_VAL_FLAG

Description: The *QT_CLEAR_FLAG* ioctl command is used to clear timer/counter flag(s). The permissible flags are Timer Compare Flag (TCF) - Bit 15, Timer Overflow Flag (TOF) - Bit 13, Input Edge Flag (IEF) - Bit 11 and Forced OFLAG Value flag (VAL) - Bit 3. All of them are located in the timer/counter Status and Control Register (TMRSCR).

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_CLEAR_FLAG* ioctl command is implemented as a macro.

Example 5-649. QT_CLEAR_FLAG

```
ioctl(QTIMER_A0, QT_CLEAR_FLAG, QT_COMPARE_FLAG |
QT_OVERFLOW_FLAG | QT_INPUT_EDGE_FLAG);
```

This code clears all three specified flags in the Status and Control Register (TMRSCR) of timer/counter A0.

Example 5-650. QT_CLEAR_FLAG

```
ioctl(QTIMER_A0, QT_CLEAR_FLAG, QT_INPUT_EDGE_FLAG);
```

This code clears the Input Edge Flag (IEF) in the Status and Control Register of timer/counter A0.

5.20.3.11 QT_CLEAR_COMPARE_FLAG - clear timer/counter compare flags

Call(s):

```
void ioctl(const int *pModuleBase, QT_CLEAR_COMPARE_FLAG,
           UWord16 Param);
```

Arguments:

Table 5-742. QT_CLEAR_COMPARE_FLAG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter flag. Use consolidation of these predefined constants: QT_COMPARE1_FLAG | QT_COMPARE2_FLAG

Description: The *QT_CLEAR_COMPARE_FLAG* ioctl command is used to clear timer/counter compare flag(s). The permissible flags are Timer Compare 2 Interrupt Source (TCF2) - Bit 5 and Timer Compare 1 Interrupt Source (TCF1) - Bit 4. All of them are located in the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_CLEAR_COMPARE_FLAG* ioctl command is implemented as a macro.

Example 5-651. QT_CLEAR_COMPARE_FLAG

```
ioctl(QTIMER_A0, QT_CLEAR_COMPARE_FLAG, QT_COMPARE2_FLAG);
```

This code clears the TCF2 flag in the Timer Comparator Status/Control Register of timer/counter A0.

5.20.3.12 QT_READ_FLAG - read status of timer/counter flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_FLAG,
          UWord16 Param);
```

Arguments:

Table 5-743. QT_READ_FLAG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter flag. Use consolidation of these predefined constants: QT_COMPARE_FLAG | QT_OVERFLOW_FLAG | QT_INPUT_EDGE_FLAG | QT_VAL_FLAG

Description: The *QT_READ_FLAG* ioctl command is used to read timer/counter flag(s). The permissible flags are Timer Compare Flag (TCF) - Bit 15, Timer Overflow Flag (TOF) - Bit 13, Input Edge Flag (IEF) - Bit 11 and Forced OFLAG Value flag (VAL) - Bit 3. All of them are located in the timer/counter Status and Control Register (TMRSCR).

Returns: State of the selected flags. The return value is the content of the Status and Control Register where all bits, except the ones specified as parameter Param, are cleared:

0x0000 - no flag is set

0x8000 | 0x2000 | 0x0800 | 0x0008 - if TCF | TOF | IEF | VAL is set (where | is the operator for logical OR)

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_FLAG* ioctl command is implemented as a macro.

Example 5-652. QT_READ_FLAG

```
UWord16 uStatus;
uStatus = ioctl(QTIMER_A0, QT_READ_FLAG, QT_INPUT_EDGE_FLAG);

if (uStatus)
{
     // Input Edge Flag is set
}
else
{
     // Input Edge Flag is cleared
}
```

This code shows how to test whether the specified bit is cleared or set. First using ioctl() command, we retrieve the content of the Status and Control Register (TMRSCR), where all bits, except the Input Edge Flag (IEF) bit are cleared. Second, there is a test of the retrieved value with an “if else” statement.

5.20.3.13 QT_READ_COMPARE_FLAG - read timer/counter compare flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_COMPARE_FLAG,
          UWord16 Param);
```

Arguments:

Table 5-744. QT_READ_COMPARE_FLAG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter flag. Use consolidation of these predefined constants: QT_COMPARE1_FLAG | QT_COMPARE2_FLAG

Description: The *QT_READ_COMPARE_FLAG* ioctl command is used to read timer/counter compare flag(s). The permissible flags are Timer Compare 2 Interrupt Source (TCF2) - Bit 5 and Timer Compare 1 Interrupt Source (TCF1) - Bit 4. All of them are located in the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: State of the selected flags. The return value is the content of the Timer Comparator Status/Control Register where all bits, except the ones specified as parameter Param, are cleared:

0x0000 - bits are cleared

0x0020 - Timer Compare 2 Flag is set

0x0010 - Timer Compare 1 Flag is set

0x0030 - Timer Compare 1 and 2 Flags are set

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_COMPARE_FLAG* ioctl command is implemented as a macro.

Example 5-653. QT_READ_COMPARE_FLAG

```
UWord16 uStatus;
uStatus = ioctl(QTIMER_A0, QT_READ_COMPARE_FLAG,
                QT_COMPARE1_FLAG);
```

This code writes status of the TCF1 bit into the uStatus variable.

5.20.3.14 QT_INT_ENABLE - enable timer/counter interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, QT_INT_ENABLE, UWord16 Param);
```

Arguments:

Table 5-745. QT_INT_ENABLE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter interrupt(s). Use consolidation of these predefined constants: QT_COMPARE_INT | QT_COMPARE1_INT | QT_COMPARE2_INT | QT_OVERFLOW_INT | QT_INPUT_EDGE_INT

Description: The *QT_INT_ENABLE* ioctl command enables timer/counter interrupt(s). There are five timer/counter interrupts: the Timer Compare Interrupt, the Timer Compare 1 Interrupt, the Timer Compare 2 Interrupt, the Timer Overflow Interrupt and the Input Edge Interrupt. This command sets the corresponding enable interrupt flags in the timer/counter Status and Control Register (TMRSCR) and/or Timer Comparator Status/Control Registers (TMRCOMSCR). The flags are the Timer Compare Flag Interrupt Enable (TCFIE), bit 14, when QT_COMPARE_INT is used as a parameter, the Timer Overflow Flag Interrupt Enable (TOFIE), bit 12, when QT_OVERFLOW_INT is used as a parameter, the Input Edge Flag Interrupt Enable (IEFIE), bit 10, when QT_INPUT_EDGE_INT is used as a parameter, the Timer Compare 2 Interrupt Enable (TCF2EN), bit 7, when QT_COMPARE_INT2 is used as a parameter and the Timer Compare 1 Interrupt Enable (TCF1EN), bit 6, when QT_COMPARE_INT1 is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_INT_ENABLE* ioctl command is implemented as a macro.

Example 5-654. QT_INT_ENABLE

```
ioctl(QTIMER_A0, QT_INT_ENABLE, QT_COMPARE_INT | QT_OVERFLOW_INT
| QT_INPUT_EDGE_INT);
```

This code enables all three timer/counter A0 interrupts in the timer/counter Status and Control Register (TMRSCR).

5.20.3.15 QT_INT_DISABLE - disable timer/counter interrupt(s)

Call(s):

```
void ioctl(const int *pModuleBase, QT_INT_DISABLE, UWord16 Param);
```

Arguments:

Table 5-746. QT_INT_DISABLE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter interrupt(s). Use consolidation of these predefined constants: QT_COMPARE_INT | QT_COMPARE1_INT | QT_COMPARE2_INT | QT_OVERFLOW_INT | QT_INPUT_EDGE_INT

Description: The *QT_INT_DISABLE* ioctl command disables timer/counter interrupt(s). There are five timer/counter interrupts: the Timer Compare Interrupt, the Timer Compare 1 Interrupt, the Timer Compare 2 Interrupt, the Timer Overflow Interrupt and the Input Edge Interrupt. This command clears the corresponding enable interrupt flags in the timer/counter Status and Control Register (TMRSCR) and/or Timer Comparator Status/Control Registers (TMRCOMSCR). The flags are the Timer Compare Flag Interrupt Enable (TCFIE), bit 14, when QT_COMPARE_INT is used as a parameter, the Timer Overflow Flag Interrupt Enable (TOFIE), bit 12, when QT_OVERFLOW_INT is used as a parameter, the Input Edge Flag Interrupt Enable (IEFIE), bit 10, when QT_INPUT_EDGE_INT is used as a parameter, the Timer Compare 2 Interrupt Enable (TCF2EN), bit 7, when QT_COMPARE_INT2 is used as a parameter and the Timer Compare 1 Interrupt Enable (TCF1EN), bit 6, when QT_COMPARE_INT1 is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_INT_DISABLE* ioctl command is implemented as a macro.

Example 5-655. QT_INT_DISABLE

```
ioctl(QTIMER_D0, QT_INT_DISABLE,
                 QT_COMPARE_INT | QT_OVERFLOW_INT);
```

This code disables compare and overflow timer/counter D0 interrupts.

5.20.3.16 QT_SET_INPUT_POLARITY - set polarity of timer/counter input signal

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_INPUT_POLARITY,
           UWord16 Param);
```

Arguments:

Table 5-747. QT_SET_INPUT_POLARITY ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter input signal polarity. Use one of these predefined constants: QT_NORMAL_POLARITY / QT_INVERTED_POLARITY

Description: The *QT_SET_INPUT_POLARITY* ioctl command is used to set the polarity of the timer/counter input signal. This command clears the IPS bit (Bit 9) in the timer/counter Status and Control Register (TMRSCR) when QT_NORMAL_POLARITY is used as a parameter. This command sets the above mentioned bit when QT_INVERTED_POLARITY is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_SET_INPUT_POLARITY* ioctl command is implemented as a macro.

Example 5-656. QT_SET_INPUT_POLARITY

```
ioctl(QTIMER_D0, QT_SET_INPUT_POLARITY, QT_INVERTED_POLARITY);
```

This code sets inverted input signal polarity for timer/counter D0. The input signal of the timer/counter is inverted. For example, when it is set to mode “Count Rising Edges of Primary Source“, then, after application of this inverter on the input signal, the timer/counter counts falling edges of the primary source.

5.20.3.17 QT_READ_EXT_INPUT_PIN - read timer/counter external input pin

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_EXT_INPUT_PIN,
              NULL);
```

Arguments:

Table 5-748. QT_READ_EXT_INPUT_PIN ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_EXT_INPUT_PIN* ioctl command returns the status of the External Input Signal bit (INPUT), Bit 8 in the Status and Control Register (TMRSCR). This bit reflects the current state of the external input pin after application of the Input Polarity Select (IPS) bit.

Returns: The returned value is either:

0x0000 - INPUT bit is cleared

0x0100 - INPUT bit is set

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_EXT_INPUT_PIN* ioctl command is implemented as a macro.

Example 5-657. QT_READ_EXT_INPUT_PIN

```
UWord16 uExtInputPin;

uExtInputPin = ioctl(QTIMER_D0, QT_READ_EXT_INPUT_PIN, NULL);
```

After this code is executed, the variable uExtInputPin contains the value 0x0000 (INPUT bit is 0) or 0x0100 (INPUT bit is 1). Then the variable uExtInputPin can be easily tested with an “if else” statement.

5.20.3.18 QT_SET_CAPTURE_MODE - set timer/counter input capture mode

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_CAPTURE_MODE,
           UWord16 Param);
```

Arguments:

Table 5-749. QT_SET_CAPTURE_MODE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter input capture mode. Use one of these predefined constants: QT_CAPTURE_DISABLED / QT_RISING_EDGE / QT_FALLING_EDGE / QT_BOTH_EDGES

Description: The *QT_SET_CAPTURE_MODE* ioctl command sets the timer/counter input capture mode. The timer/counter input capture mode specifies the operation of the capture register (on which edges of the secondary source the capture register is loaded) as well as the operation of the input edge flag (IEF). The command writes to the timer/counter Status and Control Register (TMRSCR), Bits 7-6 (Input Capture Mode). The capture modes are: QT_CAPTURE_DISABLED - capture function is disabled and input edge flag interrupts are disabled, QT_RISING_EDGE / QT_FALLING_EDGE / QT_BOTH_EDGES - load the capture register on rising/falling/both edge(s) of the secondary input source.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_SET_CAPTURE_MODE* ioctl command is implemented as a macro.

Example 5-658. QT_SET_CAPTURE_MODE

```
ioctl(QTIMER_D0, QT_SET_CAPTURE_MODE, QT_BOTH_EDGES);
```

This code sets the timer/counter D0 input capture mode to “load the capture register on both edges of the secondary input source“.

5.20.3.19 QT_MASTER_MODE - enable/disable timer/counter master mode

Call(s):

```
void ioctl(const int *pModuleBase, QT_MASTER_MODE, UWord16 Param);
```

Arguments:

Table 5-750. QT_MASTER_MODE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the desired action. Use one of these predefined constants: QT_ENABLE / QT_DISABLE

Description: The *QT_MASTER_MODE* ioctl command enables/disables the timer/counter master mode. This command sets/clears the Master Mode bit (MSTR), Bit 5, in the timer/counter Status and Control Register (TMRSCR) when QT_ENABLE/QT_DISABLE is used as a parameter. Enabled Master Mode means that the compare event from this timer/counter is broadcasted to the other timers/counters in the same Quad Timer module. This broadcasted compare event is then used to re-initialize the other timers/counters and/or used to force the current values of the VAL bits to be written to the OFLAG outputs.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_MASTER_MODE* ioctl command is implemented as a macro.

Example 5-659. QT_MASTER_MODE

```
ioctl(QTIMER_D0, QT_MASTER_MODE, QT_ENABLE);
```

This code enables the timer/counter D0 master mode. Afterwards, the timer/counter D0 can re-initialize D1, D2 or D3 or can force the D1, D2, or D3 VAL bits to be written to the OFLAG outputs. To enable re-initialization, the D1, D2 or D3 must have set the Co-channel Initialization bit (bit 3) in the timer/counter Control Register (TMRCTRL). To enable OFLAG force, D1, D2 or D3 must have set the EEOF bit (bit 4) in the timer/counter Status and Control Register (TMRSCR).

5.20.3.20 QT_EXT_OFLAG_FORCE - enable timer/counter external OFLAG force

Call(s):

```
void ioctl(const int *pModuleBase, QT_EXT_OFLAG_FORCE,
           UWord16 Param);
```

Arguments:

Table 5-751. QT_EXT_OFLAG_FORCE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the desired action. Use one of these predefined constants: QT_ENABLE / QT_DISABLE

Description: The *QT_EXT_OFLAG_FORCE* ioctl command enables/disables external OFLAG force. External OFLAG force means that the other timer/counter from the same Quad Timer module, which is enabled as a master, can force the state of this timer/counter OFLAG output. This command sets/clears the Enable External OFLAG Force bit (EEOF), Bit 4, in the timer/counter Status and Control Register (TMRSCR) if QT_ENABLE/QT_DISABLE is used as a parameter.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_EXT_OFLAG_FORCE* ioctl command is implemented as a macro.

Example 5-660. QT_EXT_OFLAG_FORCE

```
ioctl(QTIMER_D0, QT_EXT_OFLAG_FORCE, QT_ENABLE);
```

This code enables the timer/counter D0 external OFLAG force.

5.20.3.21 QT_FORCE_OFLAG - force timer/counter OFLAG output

Call(s):

```
void ioctl(const int *pModuleBase, QT_FORCE_OFLAG, UWord16 Param);
```

Arguments:

Table 5-752. QT_FORCE_OFLAG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to be written to the OFLAG output. Use QT_ZERO or QT_ONE.

Description: The *QT_FORCE_OFLAG* ioctl command sets/clears the OFLAG output using the software triggered FORCE command. First this command writes the passed 1-bit value to the VAL bit (Bit 3). Then it writes 1 to the FORCE bit (Bit 2), what causes that the current value of the VAL bit is written to the OFLAG output. Both bits - VAL and FORCE are in the timer/counter Status and Control Register (TMRSCR).

Returns: None.

Range Issues: None.

Special Issues: Use this command only if the timer/counter is disabled.

Design/Implementation: The *QT_FORCE_OFLAG* ioctl command is implemented as a macro.

Example 5-661. QT_FORCE_OFLAG

```
ioctl(QTIMER_D0, QT_FORCE_OFLAG, QT_ONE);
```

This code sets the timer/counter OFLAG output.

5.20.3.22 QT_SET_OUTPUT_POLARITY - set timer/counter output signal polarity

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_OUTPUT_POLARITY,
           UWord16 Param);
```

Arguments:

Table 5-753. QT_SET_OUTPUT_POLARITY ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the timer/counter output signal polarity. Use one of these predefined constants: QT_NORMAL_POLARITY / QT_INVERTED_POLARITY

Description: The *QT_SET_OUTPUT_POLARITY* ioctl command is used to set the timer/counter output signal polarity. This command clears the OPS bit (Bit 1) in the timer/counter Status and Control Register (TMRSCR) when QT_NORMAL_POLARITY is used as a parameter. This command sets the above mentioned bit when QT_INVERTED_POLARITY is used as a parameter. When the OPS bit is set, the polarity of the output signal is inverted.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_SET_OUTPUT_POLARITY* ioctl command is implemented as a macro.

Example 5-662. QT_SET_OUTPUT_POLARITY

```
ioctl(QTIMER_D0, QT_SET_OUTPUT_POLARITY, QT_INVERTED_POLARITY);
```

This code sets the inverted output signal polarity for timer/counter D0. The output signal of the timer/counter is inverted.

5.20.3.23 QT_OUTPUT_ON_EXT_PIN - enable/disable timer/counter output

Call(s):

```
void ioctl(const int *pModuleBase, QT_OUTPUT_ON_EXT_PIN,
           UWord16 Param);
```

Arguments:

Table 5-754. QT_OUTPUT_ON_EXT_PIN ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to select the desired action. Use one of these predefined constants: QT_ENABLE / QT_DISABLE

Description: The *QT_OUTPUT_ON_EXT_PIN* ioctl command enables/disables the OFLAG output signal to be put on the external pin. This command sets/clears the Output Enable bit (OEN), Bit 0 in the timer/counter Status and Control Register (TMRSCR), when QT_ENABLE/QT_DISABLE is used as a parameter. See the device Data Sheet for a reference on which timers/counters the OFLAG output can be made available as the external pin function.

Returns: None.

Range Issues: None.

Special Issues: Use of this ioctl command may unintentionally result in clearing an interrupt that has occurred between the time the status register is read and written back.

Design/Implementation: The *QT_OUTPUT_ON_EXT_PIN* ioctl command is implemented as a macro.

Example 5-663. QT_OUTPUT_ON_EXT_PIN

```
ioctl(QTIMER_D2, QT_OUTPUT_ON_EXT_PIN, QT_ENABLE);
```

This code enables the OFLAG output signal of timer/counter D2 to be put on the external pin TD2.

5.20.3.24 QT_SET_LOAD_CONTROL1 - specify preload event for the Compare Registers 1

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_LOAD_CONTROL1,
           UWord16 Param);
```

Arguments:

Table 5-755. QT_SET_LOAD_CONTROL1 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to specify the preload event. Use one of these predefined constants: QT_NEVER_PRELOAD / QT_LOAD_ON_CMP1 / (load upon successful compare with the value in TMRCMP1) QT_LOAD_ON_CMP2 (load upon successful compare with the value in TMRCMP2)

Description: The *QT_SET_LOAD_CONTROL1* ioctl command controls when the TMRCMP1 is pre-loaded with the value from TMRCMPLD1. This command modifies the CL1 bits (Bits 1-0) in the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_LOAD_CONTROL1* ioctl command is implemented as a macro.

Example 5-664. QT_SET_LOAD_CONTROL1

```
ioctl(QTIMER_D0, QT_SET_LOAD_CONTROL1, QT_LOAD_ON_CMP1);
```

This code specifies that the timer/counter D0 TMRCMP1 is pre-loaded with the value from TMRCMPLD1 upon successful compare with the value in TMRCMP1.

5.20.3.25 QT_SET_LOAD_CONTROL2 - specify preload event for the Compare Registers 2

Call(s):

```
void ioctl(const int *pModuleBase, QT_SET_LOAD_CONTROL2,
           UWord16 Param);
```

Arguments:

Table 5-756. QT_SET_LOAD_CONTROL2 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Parameter to specify the preload event. Use one of these predefined constants: QT_NEVER_PRELOAD / QT_LOAD_ON_CMP1 / (load upon successful compare with the value in TMRCMP1) QT_LOAD_ON_CMP2 (load upon successful compare with the value in TMRCMP2)

Description: The *QT_SET_LOAD_CONTROL2* ioctl command controls when the TMRCMP2 is pre-loaded with the value from TMRCMPLD2. This command modifies the CL2 bits (Bits 3-2) in the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_SET_LOAD_CONTROL2* ioctl command is implemented as a macro.

Example 5-665. QT_SET_LOAD_CONTROL2

```
ioctl(QTIMER_D0, QT_SET_LOAD_CONTROL2, QT_NEVER_PRELOAD);
```

This code specifies that the timer/counter D0 TMRCMP2 is not pre-loaded.

5.20.3.26 QT_WRITE_CONTROL_REG - write to the timer/counter Control Register (TMRCTRL)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_CONTROL_REG,
           UWord16 Param);
```

Arguments:

Table 5-757. QT_WRITE_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Control Register (TMRCTRL).

Description: The *QT_WRITE_CONTROL_REG* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Control Register (TMRCTRL).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-666. QT_WRITE_CONTROL_REG

```
ioctl(QTIMER_D2, QT_WRITE_CONTROL_REG, 0x3E20);
```

This code writes 0x3E20 to the timer/counter D2 Control Register (TMRCTRL).

5.20.3.27 QT_WRITE_STATUS_CONTROL_REG - write to the timer/counter Status and Control Register (TMRSCR)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_STATUS_CONTROL_REG,
           UWord16 Param);
```

Arguments:

Table 5-758. QT_WRITE_STATUS_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Status and Control Register (TMRSCR).

Description: The *QT_WRITE_STATUS_CONTROL_REG* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Status and Control Register (TMRSCR).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_STATUS_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-667. QT_WRITE_STATUS_CONTROL_REG

```
ioctl(QTIMER_D2, QT_WRITE_STATUS_CONTROL_REG, 0x4000);
```

This code writes 0x4000 to the timer/counter D2 Status and Control Register (TMRSCR).

5.20.3.28 QT_WRITE_CMP_STATUS_CONTROL_REG - write to the Timer Comparator Status/Control Registers (TMRCOMSCR)

Call(s):

```
void ioctl(const int *pModuleBase,
                QT_WRITE_CMP_STATUS_CONTROL_REG, UWord16 Param);
```

Arguments:

Table 5-759. QT_WRITE_CMP_STATUS_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the Timer Comparator Status/Control Registers (TMRCOMSCR).

Description: The *QT_WRITE_CMP_STATUS_CONTROL_REG* ioctl command writes a 16-bit value (parameter Param) to the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_CMP_STATUS_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-668. QT_WRITE_CMP_STATUS_CONTROL_REG

```
ioctl(QTIMER_B2, QT_WRITE_CMP_STATUS_CONTROL_REG, 0x0001);
```

This code writes 0x0001 to the Timer B2 Comparator Status/Control Registers (TMRCOMSCR).

5.20.3.29 QT_WRITE_COMPARE_REG1 - write to the timer/counter Compare Register 1 (TMRCMP1)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_COMPARE_REG1,
           UWord16 Param);
```

Arguments:

Table 5-760. QT_WRITE_COMPARE_REG1 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Compare Register 1 (TMRCMP1).

Description: The *QT_WRITE_COMPARE_REG1* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Compare Register 1 (TMRCMP1).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_COMPARE_REG1* ioctl command is implemented as a macro.

Example 5-669. QT_WRITE_COMPARE_REG1

```
ioctl(QTIMER_D2, QT_WRITE_COMPARE_REG1, 6502);
```

This code writes 6502 to the timer/counter D2 Compare Register 1 (TMRCMP1).

5.20.3.30 QT_WRITE_COMPARE_REG2 - write to the timer/counter Compare Register 2 (TMRCMP2)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_COMPARE_REG2,
           UWord16 Param);
```

Arguments:

Table 5-761. QT_WRITE_COMPARE_REG2 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Compare Register 2 (TMRCMP2).

Description: The *QT_WRITE_COMPARE_REG2* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Compare Register 2 (TMRCMP2).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_COMPARE_REG2* ioctl command is implemented as a macro.

Example 5-670. QT_WRITE_COMPARE_REG2

```
ioctl(QTIMER_D2, QT_WRITE_COMPARE_REG2, 65002U);
```

This code writes 65002 to the timer/counter D2 Compare Register 2 (TMRCMP2).

5.20.3.31 QT_WRITE_PRELOAD_COMPARE_REG1 - write to the Timer Comparator Load Registers 1 (TMRCMPLD1)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_PRELOAD_COMPARE_REG1,
           UWord16 Param);
```

Arguments:

Table 5-762. QT_WRITE_PRELOAD_COMPARE_REG1 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the Timer Comparator Load Registers 1 (TMRCMPLD1).

Description: The *QT_WRITE_PRELOAD_COMPARE_REG1* ioctl command writes a 16-bit value (parameter Param) to the Timer Comparator Load Registers 1 (TMRCMPLD1).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_PRELOAD_COMPARE_REG1* ioctl command is implemented as a macro.

Example 5-671. QT_WRITE_PRELOAD_COMPARE_REG1

```
ioctl(QTIMER_A2, QT_WRITE_PRELOAD_COMPARE_REG1, 0x1234);
```

This code writes 0x1234 to the Timer A2 Comparator Load Registers 1 (TMRCMPLD1).

5.20.3.32 QT_WRITE_PRELOAD_COMPARE_REG2 - write to the Timer Comparator Load Registers 2 (TMRCMPLD2)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_PRELOAD_COMPARE_REG2,
           UWord16 Param);
```

Arguments:

Table 5-763. QT_WRITE_PRELOAD_COMPARE_REG2 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the Timer Comparator Load Registers 2 (TMRCMPLD2).

Description: The *QT_WRITE_PRELOAD_COMPARE_REG2* ioctl command writes a 16-bit value (parameter Param) to the Timer Comparator Load Registers 2 (TMRCMPLD2).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_PRELOAD_COMPARE_REG2* ioctl command is implemented as a macro.

Example 5-672. QT_WRITE_PRELOAD_COMPARE_REG2

```
ioctl(QTIMER_D2, QT_WRITE_PRELOAD_COMPARE_REG2, 0x5678);
```

This code writes 0x5678 to the Timer D2 Comparator Load Registers 2 (TMRCMPLD2).

5.20.3.33 QT_WRITE_LOAD_REG - write to the timer/counter Load Register (TMRLOAD)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_LOAD_REG,
           UWord16 Param);
```

Arguments:

Table 5-764. QT_WRITE_LOAD_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Load Register (TMRLOAD).

Description: The *QT_WRITE_LOAD_REG* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Load Register (TMRLOAD).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_LOAD_REG* ioctl command is implemented as a macro.

Example 5-673. QT_WRITE_LOAD_REG

```
ioctl(QTIMER_D2, QT_WRITE_LOAD_REG, 0x8000);
```

This code writes 0x8000 to the timer/counter D2 Load Register (TMRLOAD).

5.20.3.34 QT_WRITE_COUNTER_REG - write to the timer/counter Counter Register (TMRCNTR)

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_COUNTER_REG,
           UWord16 Param);
```

Arguments:

Table 5-765. QT_WRITE_COUNTER_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Value to write to the timer/counter Counter Register (TMRCNTR).

Description: The *QT_WRITE_COUNTER_REG* ioctl command writes a 16-bit value (parameter Param) to the timer/counter Counter Register (TMRCNTR).

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_WRITE_COUNTER_REG* ioctl command is implemented as a macro.

Example 5-674. QT_WRITE_COUNTER_REG

```
ioctl(QTIMER_D2, QT_WRITE_COUNTER_REG, 0x0000);
```

This code writes 0x0000 to the timer/counter D2 Counter Register (TMRCNTR).

5.20.3.35 QT_READ_CONTROL_REG - read the timer/counter Control Register (TMRCTRL)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_CONTROL_REG, NULL);
```

Arguments:

Table 5-766. QT_READ_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_CONTROL_REG* ioctl command returns the content of the timer/counter Control Register (TMRCTRL).

Returns: Content of the timer/counter Control Register (TMRCTRL).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-675. QT_READ_CONTROL_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_CONTROL_REG, NULL);
```

This code stores the content of the timer/counter D2 Control Register (TMRCTRL) in variable uRegVal.

5.20.3.36 QT_READ_STATUS_CONTROL_REG - read the timer/counter Status and Control Register (TMRSCR)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_STATUS_CONTROL_REG,
          NULL);
```

Arguments:

Table 5-767. QT_READ_STATUS_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_STATUS_CONTROL_REG* ioctl command returns the content of the timer/counter Status and Control Register (TMRSCR).

Returns: Content of the timer/counter Status and Control Register (TMRSCR).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_STATUS_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-676. QT_READ_STATUS_CONTROL_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_STATUS_CONTROL_REG, NULL);
```

This code stores the content of the timer/counter D2 Status and Control Register (TMRSCR) in variable uRegVal.

5.20.3.37 QT_READ_CMP_STATUS_CONTROL_REG - read the Timer Comparator Status/Control Registers (TMRCOMSCR)

Call(s):

```
UWord16 ioctl(const int *pModuleBase,
              QT_READ_CMP_STATUS_CONTROL_REG, NULL);
```

Arguments:

Table 5-768. QT_READ_CMP_STATUS_CONTROL_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_CMP_STATUS_CONTROL_REG* ioctl command returns the content of the Timer Comparator Status/Control Registers (TMRCOMSCR).

Returns: Content of the Timer Comparator Status/Control Registers (TMRCOMSCR).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_CMP_STATUS_CONTROL_REG* ioctl command is implemented as a macro.

Example 5-677. QT_READ_CMP_STATUS_CONTROL_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_CMP_STATUS_CONTROL_REG, NULL);
```

This code stores the content of the Timer D2 Comparator Status/Control Registers (TMRCOMSCR) in variable uRegVal.

5.20.3.38 QT_READ_COMPARE_REG1 - read the timer/counter Compare Register 1 (TMRCMP1)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_COMPARE_REG1,
              NULL);
```

Arguments:

Table 5-769. QT_READ_COMPARE_REG1 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_COMPARE_REG1* ioctl command returns the content of the timer/counter Compare Register 1 (TMRCMP1).

Returns: Content of the timer/counter Compare Register 1 (TMRCMP1).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_COMPARE_REG1* ioctl command is implemented as a macro.

Example 5-678. QT_READ_COMPARE_REG1

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_COMPARE_REG1, NULL);
```

This code stores the content of the timer/counter D2 Compare Register 1 (TMRCMP1) in variable uRegVal.

5.20.3.39 QT_READ_COMPARE_REG2 - read the timer/counter Compare Register 2 (TMRCMP2)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_COMPARE_REG2,
              NULL);
```

Arguments:

Table 5-770. QT_READ_COMPARE_REG2 ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_COMPARE_REG2* ioctl command returns the content of the timer/counter Compare Register 2 (TMRCMP2).

Returns: Content of the timer/counter Compare Register 2 (TMRCMP2).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_COMPARE_REG2* ioctl command is implemented as a macro.

Example 5-679. QT_READ_COMPARE_REG2

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_COMPARE_REG2, NULL);
```

This code stores the content of the timer/counter D2 Compare Register 2 (TMRCMP2) in variable uRegVal.

5.20.3.40 QT_READ_LOAD_REG - read the timer/counter Load Register (TMRLOAD)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_LOAD_REG, NULL);
```

Arguments:

Table 5-771. QT_READ_LOAD_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_LOAD_REG* ioctl command returns the content of the timer/counter Load Register (TMRLOAD).

Returns: Content of the timer/counter Load Register (TMRLOAD).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_LOAD_REG* ioctl command is implemented as a macro.

Example 5-680. QT_READ_LOAD_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_LOAD_REG, NULL);
```

This code stores the content of the timer/counter D2 Load Register (TMRLOAD) in variable uRegVal.

5.20.3.41 QT_READ_COUNTER_REG - read the timer/counter Counter Register (TMRCNTR)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_COUNTER_REG, NULL);
```

Arguments:

Table 5-772. QT_READ_COUNTER_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_COUNTER_REG* ioctl command returns the content of the timer/counter Counter Register (TMRCNTR).

Returns: Content of the timer/counter Counter Register (TMRCNTR).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_COUNTER_REG* ioctl command is implemented as a macro.

Example 5-681. QT_READ_COUNTER_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_COUNTER_REG, NULL);
```

This code stores the content of the timer/counter D2 Counter Register (TMRCNTR) in variable uRegVal.

5.20.3.42 QT_READ_CAPTURE_REG - read the timer/counter Capture Register (TMRCAP)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_CAPTURE_REG, NULL);
```

Arguments:

Table 5-773. QT_READ_COUNTER_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_CAPTURE_REG* ioctl command returns the content of the timer/counter Capture Register (TMRCAP).

Returns: Content of the timer/counter Capture Register (TMRCAP).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_CAPTURE_REG* ioctl command is implemented as a macro.

Example 5-682. QT_READ_CAPTURE_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_CAPTURE_REG, NULL);
```

This code stores the content of the timer/counter D2 Capture Register (TMRCAP) in variable uRegVal.

5.20.3.43 QT_READ_HOLD_REG - read the timer/counter Hold Register (TMRHOLD)

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_HOLD_REG, NULL);
```

Arguments:

Table 5-774. QT_READ_HOLD_REG ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_HOLD_REG* ioctl command returns the content of the timer/counter Hold Register (TMRHOLD).

Returns: Content of the timer/counter Hold Register (TMRHOLD).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *QT_READ_HOLD_REG* ioctl command is implemented as a macro.

Example 5-683. QT_READ_HOLD_REG

```
UWord16 uRegVal;

uRegVal = ioctl(QTIMER_D2, QT_READ_HOLD_REG, NULL);
```

This code stores the content of the timer/counter D2 Hold Register (TMRHOLD) in variable uRegVal.

5.20.3.44 QT0_MASS_ENABLE - simultaneous count-enable of timer channels

Call(s):

```
void ioctl(const int *pModuleBase, QT0_MASS_ENABLE,
           UWord16 Param);
```

Arguments:

Table 5-775. QT0_MASS_ENABLE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Use any combination of the following parameters (one for each timer channel): QT_CH0 | QT_CH1 | QT_CH2 | QT_CH3

Description: The *QT0_MASS_ENABLE* ioctl command uses the Channel Enable Register (ENBL) to enable counting of multiple timer channels simultaneously. Note that all four timer channels are enabled by default so to achieve such a simultaneous start, the channels need to be disabled before initializing them.

Note that this command is prefixed with QT0 to remind that it can be applied only to the channel 0 of a timer module where channels are to be enabled simultaneously.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x. It can be applied only to channel 0 of a timer module.

Design/Implementation: The *QT0_MASS_ENABLE* ioctl command is implemented as a macro.

Example 5-684. QT0_MASS_ENABLE

```
ioctl(QTIMER_A0, QT_INIT, NULL);
ioctl(QTIMER_A1, QT_INIT, NULL);
ioctl(QTIMER_A2, QT_INIT, NULL);

...

ioctl(QTIMER_A0, QT0_MASS_ENABLE, QT_CH0 | QT_CH1 | QT_CH2);
```

This code enables the channels 0, 1 and 2 of the timer A simultaneously, after they are all initialized. Note that you have to explicitly disable the channels in the static configuration (Graphical Configuration Tool) so they are not started after the QT_INIT ioctl command.

5.20.3.45 QT0_MASS_DISABLE - disable timer channels

Call(s):

```
void ioctl(const int *pModuleBase, QT0_MASS_DISABLE,
           UWord16 Param);
```

Arguments:

Table 5-776. QT0_MASS_DISABLE ioctl call arguments

pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
Param	in	Use any combination of the following parameters (one for each timer channel): QT_CH0 | QT_CH1 | QT_CH2 | QT_CH3

Description: The *QT0_MASS_DISABLE* ioctl command disables counting of the selected timer channels in the Channel Enable Register (ENBL). See the QT0_MASS_ENABLE ioctl command for more details about how these commands can be used to enable counting on multiple timer channels simultaneously.

Note that this command is prefixed with QT0 to remind that it can be applied only to the channel 0 of a timer module where channels are to be disabled.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x. It can be applied only to channel 0 of a timer module.

Design/Implementation: The *QT0_MASS_DISABLE* ioctl command is implemented as a macro.

Example 5-685. QT0_MASS_DISABLE

```
ioctl(QTIMER_A0, QT0_MASS_DISABLE, QT_CH0 | QT_CH1 | QT_CH2);
```

This code disables channels 0, 1 and 2 of the timer A simultaneously.

5.20.3.46 QT_WRITE_FILT_REG - write the Input Filter Register

Call(s):

```
void ioctl(const int *pModuleBase, QT_WRITE_FILT_REG,
     UWord16 param);
```

Arguments:

Table 5-777. QT_WRITE_FILT_REG ioctl call arguments

*pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.
param	in	UWord16 register value.

Description: The *QT_WRITE_FILT_REG* ioctl command writes directly to the Input Filter Register. On MC56F802x/3x, the timer input signal can be filtered by applying a digital "counting" filter. The FILT register contains all necessary bits and bit-fields to configure the filter so this command may be used to reconfigure filtering feature in run-time. However, the input filter is typically configured statically (e.g. by using the Graphical Configuration Tool).

This command writes the parameter value directly to the Input Filter Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *QT_WRITE_FILT_REG* ioctl command is implemented as a macro.

Example 5-686. QT_WRITE_FILT_REG

```
ioctl(QTIMER_A0, QT_WRITE_FILT_REG, 0x0501);
```

This code configures the input filter in terms that the glitches shorter than 8 clocks are filtered-out. On the other hand, this filter causes 10 clocks latency for the timer input signal.

5.20.3.47 QT_READ_FILT_REG - read the Input Filter Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, QT_READ_FILT_REG, NULL);
```

Arguments:

Table 5-778. QT_READ_FILT_REG ioctl call arguments

*pModuleBase	in	The timer/counter identifier. Use QTIMER_xx, where xx can be A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3 or 0, 1, 2, 3 on devices with only one timer module. Note that not all QTIMER_xx modules are available on all chips.

Description: The *QT_READ_FILT_REG* ioctl command reads and returns directly the value of the Input Filter Register.

Returns: UWord16 register value.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *QT_READ_FILT_REG* ioctl command is implemented as a macro.

Example 5-687. QT_READ_FILT_REG

```
UWord16 filt = ioctl(QTIMER_A0, QT_READ_FILT_REG, NULL);
ioctl(QTIMER_B2, QT_WRITE_FILT_REG, filt);
```

This code copies the filter settings from Timer A Channel 0 to Timer B Channel 2.

5.20.4 Quad Timer Driver Application

The Quad Timer driver application is designed for Motorola/Freescale EVMs and is intended to illustrate the usage of this driver by a real example and also to verify the functionality by a blinking LED.

The timer/counter D2 is programmed to generate a periodic interrupt every 100 ms. The timer/counter is initialized statically, using ioctl command QT_INIT, which configures the timer/counter with data from *appconfig.h*. Inside the timer/counter D2 interrupt service routine the state of the debugging LED is changed to indicate proper function. Note that it is the user's responsibility to clear the interrupt flag inside the interrupt service routine.

The Quad Timer driver application can be found at e.g. {DSP56800E_Quick_Start Source}\..*sample_applications\MC56F8346EVM\qtimer_demo* and consists of the application project *qtimer_demo.mcp* and the source code for the application *main.c*.

Example 5-688. Quad Timer driver application - appconfig.h

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: file for static configuration of the application
*              (initial values, interrupt vectors)
*
*****************************************************************************/

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*.************************************************************************
*
*  File generated by Graphical Configuration Tool Fri, 09/Feb/2007, 14:05:22
*
************************************************************************.*/

#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x0203000fL

/*.
    OCCS Configuration
----------------------------------------
    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP Runs in Stop Mode: Disable
    COP Runs in Wait Mode: Disable
    COP Write Protect: Disable
.*/
```

```
#define OCCS_PLLCR_INIT                  0x0082
#define OCCS_PLLDB_INIT                  0x201D

/*.
    SYS Configuration
-----------------------------------------
    SIM: Power Saving Modes: Stop enabled , Wait enabled
         OnCE clock to processor core: Enabled when core TAP enabled
    Clock Output Mode: Off: Tristated
    SIM - Interrupts: Low voltage 2.2V: Disable
                      Low voltage 2.7V: Disable
    SIM - Pull-up disabled: CAN: No
                            Control Bus: No ,
                            EMI_MODE: No , JTAG: No
                            PWM A0: No , PWM A1: No
                            RESETB: No
                            XBOOT: No , IRQ: No
    SIM - Peripheral clock: PWM A: Enable , PWM B: Enable , SPI 0: Enable
                            SPI 1: Enable , SCI 0: Enable , SCI 1: Enable
                            TMR A: Enable , TMR B: Enable , TMR C: Enable
                            TMR D: Enable , DEC 0: Enable , DEC 1: Enable
                            CAN: Enable , ADC A: Enable , ADC B: Enable
                            EMI: Enable
.*/
#define SIM_GPS_INIT                     0x0000

/*.
    SEMI Configuration
-----------------------------------------
    Ext. bus driven when inactive : Disable
    Base (no CS) Write Wait States: 23
    Base (no CS) Read  Wait States: 23
    Minimal Delay before CS access: 0
    Chip Select CS0: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Both
bytes enable
                     R/W: Read / Write , PS/DS select: PS only
    Chip Select CS1: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Lower
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS2: Base address: 0x0, Blocksize: 128K , Byte Enable: 128K: Upper
byte enable
                     R/W: Read / Write , PS/DS select: DS only
    Chip Select CS3: Base address: 0x0, Blocksize: 32K , Byte Enable: 32K: Disable
                     R/W: Disable , PS/DS select: Disable
    Wait States CS0: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS1: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS2: Read  Wait States: 3, CS Setup: 0, CS Hold: 0
                     Write Wait States: 3, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
    Wait States CS3: Read  Wait States: 23, CS Setup: 0, CS Hold: 0
                     Write Wait States: 23, CS Setup: 0, CS Hold: 0
                     Minimal Delay before other CS access: 3
.*/
#define SEMI_CSBAR0_INIT                 0x0005
#define SEMI_CSBAR1_INIT                 0x0005
#define SEMI_CSBAR2_INIT                 0x0005
#define SEMI_CSOR0_INIT                  0x1FC3
#define SEMI_CSOR1_INIT                  0x1BA3
#define SEMI_CSOR2_INIT                  0x1DA3

/*.
    INTC Configuration
```

```
-----------------------------------------
    All maskable interrupts disabled: No
    IRQ A trigger mode: Low-level sensitive
    IRQ B trigger mode: Low-level sensitive
.*/
#define INTC_ICTL_INIT                  0x0000
#define INT_VECTOR_ADDR_58              qt_isr
#define INT_PRIORITY_LEVEL_58           INTC_LEVEL0

/*.
    QT_C2 Configuration
-----------------------------------------
    Count mode: Counting mode, count rising edges of primary source
    Primary source: Prescaler (IPB clock/ 128) ,
    Input polarity: True , Output polarity: True
    Input capture mode: Capture disabled, Input Edge Flag disabled
    Output mode: Asserted while counter is active
    Count stop mode: Count repeatedly , Count length: Roll over , Count direction:
Count up
    Force OFLAG output at startup: No , Forced OFLAG value: 0
    Master mode (broadcast compare event): Disable
    Enable external OFLAG force (on broadcasted event): No
    Co-channel initialization (on broadcasted event): No
    Preload Control: 1: Never , Load Reg: 0
                     2: Never , Load Reg: 0
    Interrupts: Overflow: Enabled
                Input edge: Disabled
                Compare: Enabled
                Cmp 1: Disabled
                Cmp 2: Disabled
.*/
#define QT_C2_CTRL_INIT                 0x3E00
#define QT_C2_SCR_INIT                  0x5000
#define QT_C2_CMP1_INIT                 0x8000

/*.
    GPIO_C Configuration
-----------------------------------------
    Pin  0: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  1: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  2: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  3: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  4: Function: PHASEA0/TA0 , PullUp: Enable ,
    Pin  5: Function: PHASEB0/TA1 , PullUp: Enable ,
    Pin  6: Function: INDEX0/TA2 , PullUp: Enable ,
    Pin  7: Function: HOME0/TA3 , PullUp: Enable ,
    Pin  8: Function: ISA0 , PullUp: Enable ,
    Pin  9: Function: ISA1 , PullUp: Enable ,
    Pin 10: Function: ISA2 , PullUp: Enable ,
.*/
#define GPIO_C_DDR_INIT                 0x000F
#define GPIO_C_PER_INIT                 0x07F0

/*.
    GPIO_D Configuration
-----------------------------------------
    Pin  0: Function: CS2 , PullUp: Enable ,
    Pin  1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
Int.Polarity: Active high ,
    Pin  6: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
```

```
    Pin  7: Function: GPIO , Direction: Output , Init.Value: Low - 0 , Interrupt:
Disable, Int.Polarity: Active high , Output-Mode: Push-pull ,
    Pin  8: Function: PS/CS0 , PullUp: Enable ,
    Pin  9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_DDR_INIT                 0x00C0
#define GPIO_D_PER_INIT                 0x1F01


/*.         End of autogenerated code
********************************************************************** ..*/

#endif
```

Example 5-689. Quad Timer driver application - main.c

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004-2007 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Sample application demonstrating usage of Timer driver.
*   One timer channel is configured in appconfig.h to generate two periodic
*   interrupts, one at compare event and one on consecutive rollover event.
*   Both interrupts share the same interrupt vector and service routine
*   which toggles yellow LED for compare and red LED for rollover events.
*
*
* TARGET: MC56F8xxx devices (different timer used for 56F80xx and 56F83xx see board.h)
*
*****************************************************************************/

#include "qs.h"

#include "cop.h"
#include "sys.h"
#include "intc.h"
#include "gpio.h"
#include "qtimer.h"

/* board-specific LEDs and device-specific timer module */
#include "../board.h"

/* forward prototypes */
void device_init(void);

/*
 *  The main
 */

void main (void)
{
    /* initialize SYS, COP and pins */
    device_init();

    /* Initialization of used timer according to configurable
       items defined in appconfig.h. See appconfig.h and GCT. */
    ioctl(QTIMER_DEMO, QT_INIT, NULL);

    /* initialize interrupt controler */
    ioctl(INTC, INTC_INIT, NULL);

    /* clear LEDs and pending timer interrupt flags */
    ioctl(GPIO_LEDS, GPIO_CLEAR_PIN, LED_R | LED_Y);
    ioctl(QTIMER_DEMO, QT_CLEAR_FLAG, QT_COMPARE_FLAG | QT_OVERFLOW_FLAG);

    /* enable interrupts in Status Register (SR), bits I1 and I0 */
    archEnableInt();

    /* endless loop */
    while (1)
    {
        /* keep the watchdog counter clear (in case it is enabled) */
        ioctl(COP, COP_CLEAR_COUNTER, NULL);
```

```
    }
}

/*
 *   Timer Compare Interrupt Service Routine
 */

#pragma interrupt on

void qt_isr(void)
{
    if(ioctl(QTIMER_DEMO, QT_READ_FLAG, QT_COMPARE_FLAG))
    {
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_Y);
        ioctl(QTIMER_DEMO, QT_CLEAR_FLAG, QT_COMPARE_FLAG);
    }

    if(ioctl(QTIMER_DEMO, QT_READ_FLAG, QT_OVERFLOW_FLAG))
    {
        ioctl(GPIO_LEDS, GPIO_TOGGLE_PIN, LED_R);
        ioctl(QTIMER_DEMO, QT_CLEAR_FLAG, QT_OVERFLOW_FLAG);
    }
}

#pragma interrupt off


/*
 *  Initialize SYS and all GPIO modules.
 */

void device_init(void)
{
    ioctl(SYS, SYS_INIT, NULL);
    ioctl(COP, COP_INIT, NULL);

    /* Note: This code is targeted to all 56F8xxx-based evaluation boards.
       So we need to check what GPIO instances are actually implemented */
    #ifdef GPIO_A
    ioctl(GPIO_A, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_B
    ioctl(GPIO_B, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_C
    ioctl(GPIO_C, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_D
    ioctl(GPIO_D, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_E
    ioctl(GPIO_E, GPIO_INIT, NULL);
    #endif
    #ifdef GPIO_F
    ioctl(GPIO_F, GPIO_INIT, NULL);
    #endif
}
```

5.21 PIT Driver

This section describes the API of MC56F802x/3x and MC56F800x Periodic Interrupt Timer (PIT) on-chip module. The functionality of the PIT module itself is described in the **56F802x/3x Peripheral Reference Manual** and **56F800x Peripheral Reference Manual**.

5.21.1 Introduction

The MC56F802x/3x devices have up to three Periodic Interval Timer modules and MC56F800x devices have one Periodic Interval Timer module. Each PIT module contains a 16-bit up counter, a modulo register, and a control register. The modulo and control registers are read/writable. The counter is read only. The modulo register is loaded with a value to count to and the pre-scaler is set to determine the counting rate. When enabled, the counter counts up to the modulo value and sets a flag (and an interrupt request if enabled), then it resets to $0000, and resumes counting.

This section describes the PIT driver software, providing the low level software layer interfacing hardware with software.

5.21.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.21.3.

Table 5-779. PIT Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F8023	MC56F802x/3x	MC56F83xx
PIT (PIT_BASE)	0xF2E0	N/A	N/A	N/A	N/A
PIT_0 (PIT0_BASE)	N/A	N/A	0xF190	0xF190	N/A
PIT_1 (PIT1_BASE)	N/A	N/A	N/A	0xF1A0	N/A
PIT_2 (PIT2_BASE)	N/A	N/A	N/A	0xF1B0	N/A

5.21.2.1 API Definition

The following header files are needed in order to use the PIT device driver:

Required Header File(s):

```
#include "qs.h"
#include "pit.h"
```

The following information may be found in the header file *pit.h*.

Public Data Structure(s):

None

5.21.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static PIT_0, PIT_1, PIT_2 and PIT module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-780. PIT Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
PIT_x_CTRL_INIT	UWord16	The initial value of the PIT Control Register
PIT_x_MOD_INIT	UWord16	Initial value of the PIT Modulo Register

Note - PIT_*x* should be replaced by PIT_0 for the PIT 0 module, PIT_1 for the PIT 1 module and PIT_2 for the PIT 2 module or by the PIT on MC56F800x. Note that some timers may not be available on all devices.

5.21.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call “changes” the PIT device modes or accesses the PIT register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-781. PIT Driver Arguments - ioctl

pModuleBase	in	PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.
cmd	in	Commands found in *pit.h* which are used to modify the PIT module status and control registers. See Table 5-782.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-782. ioctl commands

cmd	param	Return	Description
PIT_INIT	NULL	None	Initializes PIT module by data from the configuration file (appconfig.h).
PIT_COUNTER	PIT_ENABLE/ PIT_DISABLE	None	Enables or disables PIT counter.
only on MC56F802x/3x: PIT_SLAVE_MODE	PIT_ENABLE/ PIT_DISABLE	None	Enables or disables PIT to be controlled simultaneously with PIT_0.
PIT_ROLLOVER_INT	PIT_ENABLE/ PIT_DISABLE	None	Enables or disables the PIT interrupt to be generated.
PIT_CLEAR_ROLLOVER_INT	NULL	None	Clears interrupt flag.
PIT_SET_PRESCALER	PIT_PRESCALER_1 / PIT_PRESCALER_2 / PIT_PRESCALER_4 / PIT_PRESCALER_8 / PIT_PRESCALER_16 / PIT_PRESCALER_32 / PIT_PRESCALER_64 / PIT_PRESCALER_128 / PIT_PRESCALER_256 / PIT_PRESCALER_512 / PIT_PRESCALER_1024 / PIT_PRESCALER_2048 / PIT_PRESCALER_4096 / PIT_PRESCALER_8192 / PIT_PRESCALER_16384 / PIT_PRESCALER_32768	None	Sets PIT counter prescaler.
PIT_WRITE_MODULO_REG	UWord16	None	Sets PIT modulo value.

Table 5-782. ioctl commands (Continued)

cmd	param	Return	Description
PIT_READ_MODULO_REG	NULL	UWord16	Reads PIT modulo value.
PIT_READ_COUNTER_REG	NULL	UWord16	Reads PIT immediate counter value.

5.21.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.21.3.1 PIT_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, PIT_INIT, NULL);
```

Arguments:

Table 5-783. PIT_INIT ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.

Description: The *PIT_INIT* ioctl command executes the initialization function, which initializes all configured PIT registers by the values defined in *appconfig.h*. It is intended for static configuration of the PIT module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_INIT* ioctl command is implemented as a function call. Only necessary PIT registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-690. PIT_INIT

```
ioctl(PIT_0, PIT_INIT, NULL);
ioctl(PIT_1, PIT_INIT, NULL);
ioctl(PIT_2, PIT_INIT, NULL);
```

This code initializes the PIT modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.21.3.2 PIT_COUNTER - enable or disable PIT counter

Call(s):

```
void ioctl(const int *pModuleBase, PIT_COUNTER, UWord16 param);
```

Arguments:

Table 5-784. PIT_COUNTER ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.
param	in	Use PIT_ENABLE to enable or PIT_DISABLE to disable PIT counter.

Description: The *PIT_COUNTER* ioctl command enables (PIT_ENABLE) or disables (PIT_DISABLE) the PIT module counter. This command has no effect on PIT_1 or PIT_2 modules if these operate in “slave mode”. In “slave mode” the PIT module counter is enabled or disabled simultaneously with PIT_0 counter. See the PIT_SLAVE_MODE command for more details.

This command writes directly into the Count Enable (CNT_EN) bit of the PIT Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_COUNTER* command is implemented as a macro.

Example 5-691. PIT_COUNTER

```
ioctl(PIT_0, PIT_COUNTER, PIT_ENABLE);
```

This code enables the PIT_0 counter.

5.21.3.3 PIT_SLAVE_MODE - enable or disable PIT slave mode

Call(s):

```
void ioctl(const int *pModuleBase, PIT_SLAVE_MODE, UWord16 param);
```

Arguments:

Table 5-785. PIT_SLAVE_MODE ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_1 or PIT_2. This command has no effect on PIT_0.
param	in	Use PIT_ENABLE to enable or PIT_DISABLE to disable PIT slave mode.

Description: The *PIT_SLAVE_MODE* ioctl command enables (PIT_ENABLE) or disables (PIT_DISABLE) the PIT "slave mode" and is applicable to PIT_1 and PIT_2 only. When slave mode is enabled on PIT, enabling or disabling PIT_0 counter also enables or disables this PIT. Slave mode can be used for simultaneous count-enabling or disabling of multiple PIT modules.

This command writes directly into the Slave (SLAVE) bit of PIT Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *PIT_SLAVE_MODE* command is implemented as a macro.

Example 5-692. PIT_SLAVE_MODE

```
ioctl(PIT_1, PIT_SLAVE_MODE, PIT_ENABLE);
...
ioctl(PIT_0, PIT_COUNTER, PIT_ENABLE);
```

This code puts the PIT_1 module into the "slave mode" and then enables PIT_0, which causes PIT_1 to be enabled simultaneously.

5.21.3.4 PIT_ROLLOVER_INT - enable or disable PIT interrupt

Call(s):

```
void ioctl(const int *pModuleBase, PIT_ROLLOVER_INT,
     UWord16 param);
```

Arguments:

Table 5-786. PIT_ROLLOVER_INT ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.
param	in	Use PIT_ENABLE to enable or PIT_DISABLE to disable PIT rollover interrupt.

Description: The *PIT_ROLLOVER_INT* ioctl command enables (PIT_ENABLE) or disables (PIT_DISABLE) the PIT interrupt to be generated. The PIT interrupt is generated when PIT counter value wraps to zero after reaching the modulo value.

This command writes directly into the PIT Roll-over Interrupt Enable (PRIE) bit of the PIT Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_ROLLOVER_INT* command is implemented as a macro.

Example 5-693. PIT_ROLLOVER_INT

```
ioctl(PIT_1, PIT_ROLLOVER_INT, PIT_ENABLE);
```

This code enables the PIT roll-over interrupt to be generated on the PIT_1 module.

5.21.3.5 PIT_CLEAR_ROLLOVER_INT - clear PIT interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, PIT_CLEAR_ROLLOVER_INT, NULL);
```

Arguments:

Table 5-787. PIT_CLEAR_ROLLOVER_INT ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.

Description: The *PIT_CLEAR_ROLLOVER_INT* ioctl command clears the PIT interrupt flag. Typically, this command is used in the PIT interrupt service routine to acknowledge the interrupt event.

This command writes directly into the PIT Roll-over Flag (PRF) bit of the PIT Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_CLEAR_ROLLOVER_INT* command is implemented as a macro.

Example 5-694. PIT_CLEAR_ROLLOVER_INT

```
ioctl(PIT_1, PIT_CLEAR_ROLLOVER_INT, NULL);
```

This code clears the PIT_1 roll-over event flag.

5.21.3.6 PIT_SET_PRESCALER - set PIT counter prescaler

Call(s):

```
void ioctl(const int *pModuleBase, PIT_SET_PRESCALER,
     UWord16 param);
```

Arguments:

Table 5-788. PIT_SET_PRESCALER ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.
param	in	Use one of the predefined constants: PIT_PRESCALER_1 PIT_PRESCALER_2 PIT_PRESCALER_4 PIT_PRESCALER_8 PIT_PRESCALER_16 PIT_PRESCALER_32 PIT_PRESCALER_64 PIT_PRESCALER_128 PIT_PRESCALER_256 PIT_PRESCALER_512 PIT_PRESCALER_1024 PIT_PRESCALER_2048 PIT_PRESCALER_4096 PIT_PRESCALER_8192 PIT_PRESCALER_16384 PIT_PRESCALER_32768.

Description: The *PIT_SET_PRESCALER* ioctl command configures the counter prescaler. The PIT count clock is derived directly from the IP Bus clock, divided by the prescaler value.

This command writes directly into the Prescaler (PRESCALER) bit-field of the PIT Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_SET_PRESCALER* command is implemented as a macro.

Example 5-695. PIT_SET_PRESCALER

```
ioctl(PIT_1, PIT_SET_PRESCALER, PIT_PRESCALER_32768);
```

This code sets the maximum possible value to the PIT prescaler, causing PIT_1 increment period to be 1.024 ms at 32MHz IP Bus clock.

5.21.3.7 PIT_WRITE_MODULO_REG - set PIT counter terminal value

Call(s):

```
void ioctl(const int *pModuleBase, PIT_WRITE_MODULO_REG,
     UWord16 param);
```

Arguments:

Table 5-789. PIT_WRITE_MODULO_REG ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.
param	in	UWord16 modulo value.

Description: The *PIT_WRITE_MODULO_REG* ioctl command writes the counter modulo register, which presents the terminal value of the PIT counter. When counter reaches the modulo value, it is wrapped to zero at next counter clock and the PIT interrupt event is raised (if enabled). Modulo value of zero causes the PIT counter to remain zero and to generate the PIT interrupts at each counter clock (IP Bus clock divided by prescaler value).

This command writes directly into the Modulo-Value Register (MOD).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_WRITE_MODULO_REG* command is implemented as a macro.

Example 5-696. PIT_WRITE_MODULO_REG

```
ioctl(PIT_1, PIT_SET_PRESCALER, PIT_PRESCALER_32);
ioctl(PIT_1, PIT_WRITE_MODULO_REG, 999);
```

This code configures PIT_1 module to generate 1kHz periodic interrupts at 32MHz IP Bus Clock.

5.21.3.8 PIT_READ_MODULO_REG - read PIT counter terminal value

Call(s):

```
UWoprd16 ioctl(const int *pModuleBase, PIT_READ_MODULO_REG,
     NULL);
```

Arguments:

Table 5-790. PIT_READ_MODULO_REG ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.

Description: The *PIT_READ_MODULO_REG* ioctl command reads and returns the Modulo Value Register content.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_READ_MODULO_REG* command is implemented as a macro.

Example 5-697. PIT_READ_MODULO_REG

```
UWord16 mod;

mod = ioctl(PIT_1, PIT_READ_MODULO_REG, NULL);
ioctl(PIT_2, PIT_WRITE_MODULO_REG, mod);
```

This code copies the modulo value from PIT_1 to PIT_2.

5.21.3.9 PIT_READ_COUNTER_REG - read PIT counter value

Call(s):

```
UWord16 ioctl(const int *pModuleBase, PIT_READ_COUNTER_REG,
     NULL);
```

Arguments:

Table 5-791. PIT_READ_COUNTER_REG ioctl call arguments

*pModuleBase	in	The PIT module identifier. Use PIT_0, PIT_1, PIT_2 or PIT.

Description: The *PIT_READ_COUNTER_REG* ioctl command reads and returns the PIT Counter register content.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x and MC56F800x.

Design/Implementation: The *PIT_READ_COUNTER_REG* command is implemented as a macro.

Example 5-698. PIT_READ_COUNTER_REG

```
UWord16 count;

count = ioctl(PIT_0, PIT_READ_COUNTER_REG, NULL);
```

This code reads the immediate counter value of PIT_1.

5.21.4 PIT Driver Applications

There are three sample applications demonstrating how PIT module can be used in a user application. The supported EVM boards for PIT sample application are MC56F8025DEMO and MC56F8037EVM.

5.21.4.1 dac_pit_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\dac_pit_demo

This application shows how PIT_0 and PIT_1 are driven by PIT_0 and PIT_1 timers in an automatic mode, generating a sawtooth signals.

5.21.4.2 dac_cmp_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\dac_cmp_demo

This application shows how PIT_0 and PIT_1 are driven by PIT_0 timer in an automatic mode, generating a sawtooth signals. Analog Comparator module (CMP) is then used to compare the PIT_0 and PIT_1 signals and sets one user LED accordingly.

5.21.4.3 adc_fmstr_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\adc_fmstr_demo

Again, this application shows how PIT_0 and PIT_1 are driven by PIT_0 counter in an automatic mode, generating a sawtooth signals. The ANA7 and ANB7 channels are sourced from PIT_0 and PIT_1 and the ADC-measured values may be displayed in the FreeMASTER visualization tool.

5.22 CMP Driver

This section describes the API of MC56F802x/3x Analog Comparator (CMP) on-chip module. The functionality of the CMP module itself is described in the **56F802x/3x Peripheral Reference Manual**.

5.22.1 Introduction

The MC56F802x/3x devices have two Analog Comparator modules. Each comparator includes a five-input analog switching matrix with the continuous-time differential-input analog comparator function. The five analog inputs include three GPIO (referred to as CIN1, CIN2, and CIN3), a DAC input, and an import input from the other comparator module. The switching matrix supports the independent connection of the analog inputs to the + and – input of the analog comparator and to the comparator's export output.

This section describes the CMP driver software, providing the low level software layer interfacing hardware with software.

5.22.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.22.3.

Table 5-792. CMP Module Base Address

Module base address of / for	MC56F801x	MC56F802x/3x	MC56F83xx
CMP_A (CMP0_BASE)	N/A	0xF1E0	N/A
CMP_B (CMP1_BASE)	N/A	0xF1F0	N/A

5.22.2.1 API Definition

The following header files are needed in order to use the CMP device driver:

Required Header File(s):

```
#include "qs.h"
#include "cmp.h"
```

The following information may be found in the header file *cmp.h*.

Public Data Structure(s):

None

5.22.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static CMP_A and CMP_B module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-793. CMP Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
CMP_x_CTRL_INIT	UWord16	The initial value of the Comparator Control Register
CMP_x_FILT_INIT	UWord16	Initial value of the Comparator Filter Register

Note - The "x" specifies the CMP module. This may be either A or B.

5.22.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the CMP device modes or accesses the CMP register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-794. CMP Driver Arguments - ioctl

pModuleBase	in	CMP module identifier. Use CMP_A or CMP_B.
cmd	in	Commands found in *cmp.h* which are used to modify the CMP module status and control registers. See Table 5-795.

Table 5-794. CMP Driver Arguments - ioctl

param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-795. ioctl commands

Cmd	param	Return	Description
CMP_INIT	NULL	None	Initializes CMP module by data from configuration file (appconfig.h).
CMP_MODULE	CMP_ENABLE/ CMP_DISABLE	None	Enables or disables CMP device.
CMP_INVERT	CMP_ENABLE/ CMP_DISABLE	None	Enables logic level inverter on output.
CMP_SELECT_POS_INPUT	CMP_CIN0 / CMP_CIN1 / CMP_CIN2 / CMP_DAC / CMP_IMPORT	None	Selects the source of the positive comparator input
CMP_SELECT_NEG_INPUT	CMP_CIN0 / CMP_CIN1 / CMP_CIN2 / CMP_DAC / CMP_IMPORT	None	Selects the source of the negative comparator input
CMP_SELECT_EXPORT_OUTPUT	CMP_CIN0 / CMP_CIN1 / CMP_CIN2 / CMP_DAC	None	Selects the “EXPORT” output of the comparator. This analog output can be used as an “IMPORT” input by the other comparator module.
CMP_INT_ENABLE	CMP_RISING_EDGE | CMP_FALLING_EDGE	None	Enables comparator interrupts.
CMP_INT_DISABLE	CMP_RISING_EDGE | CMP_FALLING_EDGE	None	Disables interrupts.
CMP_READ_INT_FLAGS	NULL	UWord16	Reads interrupt pending flags.
CMP_CLEAR_INT_FLAGS	CMP_RISING_EDGE | CMP_FALLING_EDGE	None	Clears interrupt request flags.
CMP_READ_OUTPUT	NULL	UWord16	Gets actual comparator output value (can be used for a zero/non-zero testing only)

Table 5-795. ioctl commands (Continued)

Cmd	param	Return	Description
CMP_WRITE_FILT_REG	UWord16	None	Writes comparator output filter register.
CMP_READ_FILT_REG	NULL	UWord16	Reads comparator output filter register.

5.22.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.22.3.1 CMP_INIT - initialize comparator module

Call(s):

```
void ioctl(const int *pModuleBase, CMP_INIT, NULL);
```

Arguments:

Table 5-796. CMP_INIT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.

Description: The *CMP_INIT* ioctl command executes the initialization function, which initializes all configured CMP registers by the values defined in *appconfig.h*. It is intended for static configuration of the CMP module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_INIT* ioctl command is implemented as a function call. Only necessary CMP registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-699. CMP_INIT

```
ioctl(CMP_A, CMP_INIT, NULL);
ioctl(CMP_B, CMP_INIT, NULL);
```

This code initializes the CMP modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.22.3.2 CMP_MODULE - enable/disable CMP device

Call(s):

```
void ioctl(const int *pModuleBase, CMP_MODULE, UWord16 param);
```

Arguments:

Table 5-797. CMP_MODULE ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use CMP_ENABLE to enable or CMP_DISABLE to disable CMP module.

Description: The *CMP_MODULE* ioctl command enables (CMP_ENABLE) or disables (CMP_DISABLE) the CMP module. Before the CMP module is used, the CMP device should be configured (in disabled state) and then enabled. Also it may be decided to let the CMP module to be enabled already during the initialization procedure.

This command writes directly into the Power-Down (PDN) bit of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_MODULE* command is implemented as a macro.

Example 5-700. CMP_MODULE

```
/* configure desired CMP parameters */
/* ... */
/* enable it */
ioctl(CMP_A, CMP_MODULE, CMP_ENABLE);
```

This code enables the CMP_A module.

5.22.3.3 CMP_INVERT - enable/disable comparator output inverter

Call(s):

```
void ioctl(const int *pModuleBase, CMP_INVERT, UWord16 param);
```

Arguments:

Table 5-798. CMP_INVERT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use one of the predefined constants: CMP_ENABLE ... to enable output inverter CMP_DISABLE ... to disable output inverter

Description: The *CMP_INVERT* ioctl command enables or disables logic-level inverter at the comparator output.

This command writes directly into the Invert Control (INV) bit of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_INVERT* command is implemented as a macro.

Example 5-701. CMP_INVERT

```
ioctl(CMP_A, CMP_INVERT, CMP_DISABLE);
```

This code disables inverting of the CMP_A output.

5.22.3.4 CMP_SELECT_POS_INPUT - select positive comparator input

Call(s):

```
void ioctl(const int *pModuleBase, CMP_SELECT_POS_INPUT,
     UWord16 param);
```

Arguments:

Table 5-799. CMP_SELECT_POS_INPUT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use one of the predefined constants: CMP_CIN0, CMP_CIN1, CMP_CIN2 ... one of comparator input pins CMP_DAC ... DAC output (DAC_0 for CMP_A, DAC_1 for CMP_B) CMP_IMPORT ... internal “import” signal

Description: The *CMP_SELECT_POS_INPUT* ioctl selects the signal source of the comparator positive input. Similarly, the CMP_SELECT_NEG_INPUT selects signal source for the negative comparator input. The internal switching matrix is capable of selecting one of five signal sources.

CMP_CIN0-2: selected pin must be configured for comparator peripheral mode in SYS and GPIO

CMP_DAC: DAC module can be used to set the comparator level

CMP_IMPORT: selects the “import” input, which is internally connected to “export” signal of the other comparator module (there are always CMP_A and CMP_B modules). The other module selects the “export” output signal using the CMP_SELECT_EXPORT_OUTPUT command.

This command writes into the Positive Comparator Input Source Select (PSEL) bit-field of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_SELECT_POS_INPUT* command is implemented as a macro.

Example 5-702. CMP_SELECT_POS_INPUT

```
ioctl(CMP_A, CMP_SELECT_POS_INPUT, CMP_CIN2);
```

This code selects the CINA2 comparator input pin as the positive input to the CMP_A comparator unit.

5.22.3.5 CMP_SELECT_NEG_INPUT - select negative comparator input

Call(s):

```
void ioctl(const int *pModuleBase, CMP_SELECT_NEG_INPUT,
     UWord16 param);
```

Arguments:

Table 5-800. CMP_SELECT_NEG_INPUT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use one of the predefined constants: CMP_CIN0, CMP_CIN1, CMP_CIN2 ... one of comparator input pins CMP_DAC ... DAC output (DAC_0 for CMP_A, DAC_1 for CMP_B) CMP_IMPORT ... internal “import” signal

Description: The *CMP_SELECT_NEG_INPUT* ioctl selects the signal source of the comparator negative input. Similarly, the CMP_SELECT_POS_INPUT selects signal source for the positive comparator input. The internal switching matrix is capable of selecting one of five signal sources.

CMP_CIN0-2: selected pin must be configured for comparator peripheral mode in SYS and GPIO

CMP_DAC: DAC module can be used to set the comparator level

CMP_IMPORT: selects the “import” input, which is internally connected to “export” signal of the other comparator module (there are always CMP_A and CMP_B modules). The other module selects the “export” output signal using the CMP_SELECT_EXPORT_OUTPUT command.

This command writes into the Negative Comparator Input Source Select (NSEL) bit-field of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_SELECT_NEG_INPUT* command is implemented as a macro.

Example 5-703. CMP_SELECT_NEG_INPUT

```
ioctl(CMP_B, CMP_SELECT_EXPORT_OUTPUT, CMP_CIN2)
ioctl(CMP_A, CMP_SELECT_NEG_INPUT, CMP_IMPORT);
```

This code selects the CINB2 comparator input pin as the negative input to CMP_A comparator unit. This is achieved by that CINB2 signal is exported from CMP_B module and is selected in CMP_A as an imported source of negative input.

5.22.3.6 CMP_SELECT_EXPORT_OUTPUT - select internal comparator “export” output

Call(s):

```
void ioctl(const int *pModuleBase, CMP_SELECT_EXPORT_OUTPUT,
     UWord16 param);
```

Arguments:

Table 5-801. CMP_SELECT_EXPORT_OUTPUT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use one of the predefined constants: CMP_CIN0, CMP_CIN1, CMP_CIN2 ... one of comparator input pins CMP_DAC ... DAC output (DAC_0 for CMP_A, DAC_1 for CMP_B)

Description: The *CMP_SELECT_EXPORT_OUTPUT* ioctl can be used to route the selected input signal of one comparator module also to input of the other comparator module. The export output signal selected by this command can be used by the other module as the “import” input signal and used as positive or negative input to its own comparator unit. For more details, see the CMP_SELECT_POS_INPUT and CMP_SELECT_NEG_INPUT ioctl commands.

This command writes into the Export Source Select (ESEL) bit-field of CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_SELECT_EXPORT_OUTPUT* command is implemented as a macro.

Example 5-704. CMP_SELECT_EXPORT_OUTPUT

```
ioctl(CMP_B, CMP_SELECT_EXPORT_OUTPUT, CMP_CIN2)
ioctl(CMP_A, CMP_SELECT_NEG_INPUT, CMP_IMPORT);
```

This code selects the CINB2 comparator input pin as the negative input to CMP_A comparator unit. This is achieved by that CINB2 signal is exported from CMP_B module and is selected in CMP_A as an imported source of negative input.

5.22.3.7 CMP_INT_ENABLE - enable comparator interrupts

Call(s):

```
void ioctl(const int *pModuleBase, CMP_INT_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-802. CMP_INT_ENABLE ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use any combination of the following constants: CMP_RISING_EDGE ... comparator output rising edge interrupt CMP_FALLING_EDGE ... comparator output falling edge interrupt

Description: The *CMP_INT_ENABLE* ioctl command enables the selected comparator interrupts.

This command writes into the interrupt enable (RCIE and FCIE) bits of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_INT_ENABLE* command is implemented as a macro.

Example 5-705. CMP_INT_ENABLE

```
ioctl(CMP_B, CMP_INT_ENABLE, CMP_RISING_EDGE)
```

This code enables the rising-edge event interrupt of the CMP_B module. The interrupt is generated any time the comparator output goes from low to high state.

5.22.3.8 CMP_INT_DISABLE - disable comparator interrupts

Call(s):

```
void ioctl(const int *pModuleBase, CMP_INT_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-803. CMP_INT_DISABLE ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use any combination of the following constants: CMP_RISING_EDGE ... comparator output rising edge interrupt CMP_FALLING_EDGE ... comparator output falling edge interrupt

Description: The *CMP_INT_DISABLE* ioctl command disables the selected comparator interrupts.

This command writes into the interrupt enable (RCIE and FCIE) bits of the CMP Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_INT_DISABLE* command is implemented as a macro.

Example 5-706. CMP_INT_DISABLE

```
ioctl(CMP_A, CMP_INT_DISABLE, CMP_RISING_EDGE | CMP_FALLING_EDGE)
```

This code disables both interrupts of the CMP_A module.

5.22.3.9 CMP_READ_INT_FLAGS - read interrupt pending flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, CMP_READ_INT_FLAGS, NULL);
```

Arguments:

Table 5-804. CMP_READ_INT_FLAGS ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.

Description: The *CMP_READ_INT_FLAGS* ioctl command returns the value of the comparator interrupt flags. The returned value may be AND-tested for occurrence of the CMP_RISING_EDGE and/or CMP_FALLING_EDGE interrupt flags.

This command returns the RC and FC bits of the CMP Status Register.

Returns: UWord16 value of the two interrupt status bits.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_READ_INT_FLAGS* command is implemented as a macro.

Example 5-707. CMP_READ_INT_FLAGS

```
if(ioctl(CMP_A, CMP_READ_INT_FLAGS, NULL) & CMP_RISING_EDGE)
{
      ...
}
```

This code tests (polls) the rising-edge interrupt flag of the CMP_A comparator module.

5.22.3.10 CMP_CLEAR_INT_FLAGS - clear interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, CMP_CLEAR_INT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-805. CMP_CLEAR_INT_FLAGS ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	Use any combination of the following constants: CMP_RISING_EDGE ... comparator output rising edge interrupt CMP_FALLING_EDGE ... comparator output falling edge interrupt

Description: The *CMP_CLEAR_INT_FLAGS* ioctl command clears the selected interrupt flags. Typically the value returned from CMP_READ_INT_FLAGS is used as the parameter for this command in the comparator interrupt service routine.

This command writes the parameter value into the RC and FC bits of the CMP Status Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_CLEAR_INT_FLAGS* command is implemented as a macro.

Example 5-708. CMP_CLEAR_INT_FLAGS

```
UWord16 irqs = ioctl(CMP_A, CMP_READ_INT_FLAGS, NULL);

if(irqs & CMP_RISING_EDGE)
{
     ...
}

ioctl(CMP_A, CMP_CLEAR_INT_FLAGS, irqs);
```

This code shows how pending interrupts are read, processed and cleared.

5.22.3.11 CMP_READ_OUTPUT - read immediate state of comparator output

Call(s):

```
UWord16 ioctl(const int *pModuleBase, CMP_READ_OUTPUT, NULL);
```

Arguments:

Table 5-806. CMP_READ_OUTPUT ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.

Description: The *CMP_READ_OUTPUT* ioctl command returns the immediate state of comparator output as it is read from the COUT bit of the CMP Status Register. The returned value can only be tested in zero/non-zero conditions.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_READ_OUTPUT* command is implemented as a macro.

Example 5-709. CMP_READ_OUTPUT

```
if(ioctl(CMP_A, CMP_READ_OUTPUT, NULL))
{
      // output is high...
}
else
{
      // output is low...
}
```

This code shows how the immediate state of the comparator output may be tested.

5.22.3.12 CMP_WRITE_FILT_REG - write Output Filter register

Call(s):

```
void ioctl(const int *pModuleBase, CMP_WRITE_FILT_REG,
     UWord16 param);
```

Arguments:

Table 5-807. CMP_WRITE_FILT_REG ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.
param	in	UWord16 register value.

Description: The *CMP_WRITE_FILT_REG* ioctl command writes directly to the Output Filter Register. The comparator output signal can be filtered by applying a digital "counting" filter. The FILT register contain all necessary bits and bit-fields to configure the filter so this command may be used to reconfigure filtering feature in run-time. However, the output filter is typically configured statically (e.g. by using the Graphical Configuration Tool).

This command writes the parameter value directly to the Output Filter Register (FILT).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_WRITE_FILT_REG* ioctl command is implemented as a macro.

Example 5-710. CMP_WRITE_FILT_REG

```
ioctl(CMP_A, CMP_WRITE_FILT_REG, 0x0501);
```

This code configures the filter in terms that the output glitches shorter than 8 clocks are filtered-out. On the other hand, this filter causes 10 clocks latency for the comparator output signal.

5.22.3.13 CMP_READ_FILT_REG - read Output Filter register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, CMP_READ_FILT_REG, NULL);
```

Arguments:

Table 5-808. CMP_READ_FILT_REG ioctl call arguments

*pModuleBase	in	The CMP module identifier. Use CMP_A or CMP_B.

Description: The *CMP_READ_FILT_REG* ioctl command reads and returns the value of the Output Filter Register.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *CMP_READ_FILT_REG* ioctl command is implemented as a macro.

Example 5-711. CMP_READ_FILT_REG

```
UWord16 filt = ioctl(CMP_A, CMP_READ_FILT_REG, NULL);
ioctl(CMP_B, CMP_WRITE_FILT_REG, filt);
```

This code copies the output filter settings from the CMP_A to CMP_B module.

5.22.4 CMP Driver Applications

There is one sample application demonstrating how the CMP module can be used in a user application. The supported EVM boards for the CMP sample application are MC56F8025DEMO and MC56F8037EVM.

5.22.4.1 dac_cmp_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\dac_cmp_demo

This application shows how DAC_0 and DAC_1 are driven by PIT_0 timer in an automatic mode, generating a sawtooth signals. Analog Comparator module (CMP_A) is then used to compare the CMP_A and CMP_B signals and sets one user LED accordingly.

5.23 HSCMP Driver

This section describes the API of MC56F800x High Speed Analog Comparator (HSCMP) on-chip module. The functionality of the HSCMP module itself is described in the **56F800x Peripheral Reference Manual**.

5.23.1 Introduction

The MC56F800x devices have three High Speed Analog Comparator modules. Each comparator has up to eight external analog -input pins Four of these pins are positive input and four pins are negative input High speed comparator. High speed comparator output can be windowed, sampled and digitaly filtered. There is selectable interrupt on rising edge, falling edge or either rising or falling edge of comparator output. Each HSCMP output can be conected to external pin CMPx_OUT..

This section describes the HSCMP driver software, providing the low level software layer interfacing hardware with software.

5.23.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.23.3.

Table 5-809. HSCMP Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
HSCMP_0 (HSCMP0_BASE)	0xF100	N/A	N/A	N/A
HSCMP_1 (HSCMP1_BASE)	0xF110	N/A	N/A	N/A
HSCMP_2 (HSCMP2_BASE)	0xF120	N/A	N/A	N/A

5.23.2.1 API Definition

The following header files are needed in order to use the HSCMP device driver:

Required Header File(s):

```
#include "qs.h"
#include "hscmp.h"
```

The following information may be found in the header file *hscmp.h*.

Public Data Structure(s):

None

5.23.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static HSCMP_0, HSCMP_1 and HSCMP_2 module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-810. CMP Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
HSCMP_x_CR0_INIT	UWord16	The initial value of the Comparator Control Register 0
HSCMP_x_CR1_INIT	UWord16	The initial value of the Comparator Control Register 1
HSCMP_x_FPR_INIT	UWord16	Initial value of the Comparator Filter Period Register
HSCMP_x_SCR_INIT	UWord16	Initial value of the Comparator Status and Control Register

Note - The "x" specifies the HSCMP module. This may be either 0, 1 or 2.

5.23.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the HSCMP device modes or accesses the HSCMP register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-811. CMP Driver Arguments - ioctl

pModuleBase	in	HSCMP module identifier. Use HSCMP_0, HSCMP_1 or HSCMP_2.
cmd	in	Commands found in *cmp.h* which are used to modify the HSCMP module status and control registers. See Table 5-812.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-812. ioctl commands

Cmd	param	Return	Description
HSCMP_INIT	NULL	None	Initializes HSCMP module by data from configuration file (app-config.h).
HSCMP_MODULE	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables HSCMP device.
HSCMP_SELECT_POS_ INPUT	HSCMP_PIN_0 / HSCMP_PIN_1 / HSCMP_PIN_2 / HSCMP_PIN_3	None	Selects the source of the positive comparator input
HSCMP_SELECT_NEG_ INPUT	HSCMP_PIN_0 / HSCMP_PIN_1 / HSCMP_PIN_2 / HSCMP_PIN_3	None	Selects the source of the negative comparator input
HSCMP_SET_INVERT	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables logic level inverter on output.
HSCMP_SET_SAMPLE	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables sampling mode.
HSCMP_SET_WINDOWING	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables windowing mode.
HSCMP_SET_HIGH_SPEED	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables high speed mode.
HSCMP_SET_OUTPUT_PIN	HSCMP_COUNT / HSCMP_COUNTA	None	Selects comparator output source.

Table 5-812. ioctl commands (Continued)

Cmd	param	Return	Description
HSCMP_SET_OUTPUT_ ACTIVE	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables comparator output.
HSCMP_INT_RISING_EDGE	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables rising edge comparator interrupts.
HSCMP_INT_FALLING_EDGE	HSCMP_ENABLE / HSCMP_DISABLE	None	Enables or disables falling edge comparator interrupts.
HSCMP_TEST_INT_FLAGS	HSCMP_FLAG_RISING_ EDGE | HSCMP_FLAG_FALLING_ EDGE	UWord16	Reads interrupt pending flags.
HSCMP_CLEAR_INT_FLAGS	HSCMP_FLAG_RISING_ EDGE | HSCMP_FLAG_FALLING_ EDGE	None	Clears interrupt request flags.
HSCMP_TEST_OUTPUT	NULL	UWord16	Gets actual comparator output value (can be used for a zero/non-zero testing only)
HSCMP_WRITE_FILT_ COUNTER	HSCMP_FILT_COUNTER_0 / HSCMP_FILT_COUNTER_1 / HSCMP_FILT_COUNTER_2 / HSCMP_FILT_COUNTER_3 / HSCMP_FILT_COUNTER_4 / HSCMP_FILT_COUNTER_5 / HSCMP_FILT_COUNTER_6 / HSCMP_FILT_COUNTER_7	None	Writes filter sample count value.
HSCMP_READ_FILT_ COUNTER	NULL	None	Reads filter sample count value.
HSCMP_WRITE_FILT_REG	UWord16	None	Writes comparator output filter register.
HSCMP_READ_FILT_REG	NULL	UWord16	Reads comparator output filter register.

5.23.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.23.3.1 HSCMP_INIT - initialize comparator module

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_INIT, NULL);
```

Arguments:

Table 5-813. HSCMP_INIT ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.

Description: The HS*CMP_INIT* ioctl command executes the initialization function, which initializes all configured HSCMP registers by the values defined in *appconfig.h*. It is intended for static configuration of the HSCMP module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The HS*CMP_INIT* ioctl command is implemented as a function call. Only necessary HSCMP registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-712. HSCMP_INIT

```
ioctl(HSCMP_0, HSCMP_INIT, NULL);
ioctl(HSCMP_1, HSCMP_INIT, NULL);
ioctl(HSCMP_2, HSCMP_INIT, NULL);
```

This code initializes the HSCMP modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.23.3.2 HSCMP_MODULE - enable/disable HSCMP device

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_MODULE, UWord16 param);
```

Arguments:

Table 5-814. HSCMP_MODULE ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use HSCMP_ENABLE to enable or HSCMP_DISABLE to disable HSCMP module.

Description: The *HSCMP_MODULE* ioctl command enables (HSCMP_ENABLE) or disables (HSCMP_DISABLE) the HSCMP module. Before the HSCMP module is used, the HSCMP device should be configured (in disabled state) and then enabled. Also it may be decided to let the HSCMP module to be enabled already during the initialization procedure.

This command writes directly into the Comparator Module Enable (EN) bit of the HSCMP Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The HS*CMP_MODULE* command is implemented as a macro.

Example 5-713. HSCMP_MODULE

```
/* configure desired HSCMP parameters */
/* ... */
/* enable it */
ioctl(HSCMP_0, HSCMP_MODULE, HSCMP_ENABLE);
```

This code enables the HSCMP_0 module.

5.23.3.3 HSCMP_SELECT_POS_INPUT - select positive comparator input

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SELECT_POS_INPUT,
     UWord16 param);
```

Arguments:

Table 5-815. HSCMP_SELECT_POS_INPUT ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_PIN_0, HSCMP_PIN_1, HSCMP_PIN_2 ... one of comparator input pins.

Description: The *HSCMP_SELECT_POS_INPUT* ioctl selects the signal source of the comparator positive input. Similarly, the HSCMP_SELECT_NEG_INPUT selects signal source for the negative comparator input. The internal switching matrix is capable of selecting one of four signal sources.

HSCMP_PIN_0-3: selected pin must be configured for comparator peripheral mode in SYS and GPIO module.

This command writes into the Positive Comparator Input Source Select (PMC) bit-field of the HSCMP Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SELECT_POS_INPUT* command is implemented as a macro.

Example 5-714. HSCMP_SELECT_POS_INPUT

```
ioctl(HSCMP_0, HSCMP_SELECT_POS_INPUT, HSCMP_PIN_2);
```

This code selects the PIN 2 comparator input pin as the positive input to the HSCMP_0 comparator unit.

5.23.3.4 HSCMP_SELECT_NEG_INPUT - select negative comparator input

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SELECT_NEG_INPUT,
     UWord16 param);
```

Arguments:

Table 5-816. HSCMP_SELECT_NEG_INPUT ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_PIN_0, HSCMP_PIN_1, HSCMP_PIN_2 ... one of comparator input pins.

Description: The *HSCMP_SELECT_POS_INPUT* ioctl selects the signal source of the comparator positive input. Similarly, the HSCMP_SELECT_NEG_INPUT selects signal source for the negative comparator input. The internal switching matrix is capable of selecting one of four signal sources.

HSCMP_PIN_0-3: selected pin must be configured for comparator peripheral mode in SYS and GPIO module.

This command writes into the Negative Comparator Input Source Select (MMC) bit-field of the HSCMP Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SELECT_NEG_INPUT* command is implemented as a macro.

Example 5-715. HSCMP_SELECT_NEG_INPUT

```
ioctl(HSCMP_1, HSCMP_SELECT_NEG_INPUT, HSCMP_PIN_0);
```

This code selects the PIN 0 comparator input pin as the negative input to the HSCMP_1 comparator unit.

5.23.3.5 HSCMP_SET_INVERT - enable/disable comparator output inverter

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_INVERT,
           UWord16 param);
```

Arguments:

Table 5-817. HSCMP_SET_INVERT ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable output inverter HSCMP_DISABLE ... to disable output inverter

Description: The *HSCMP_SET_INVERT* ioctl command enables or disables logic-level inverter at the high speed comparator output.

This command writes directly into the Invert Control (INV) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_INVERT* command is implemented as a macro.

Example 5-716. HSCMP_SET_INVERT

```
ioctl(HSCMP_2, HSCMP_SET_INVERT, HSCMP_DISABLE);
```

This code disables inverting of the HSCMP_2 output.

5.23.3.6 HSCMP_SET_SAMPLE - enable/disable sample mode

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_SAMPLE,
           UWord16 param);
```

Arguments:

Table 5-818. HSCMP_SET_SAMPLE ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable sample mode HSCMP_DISABLE ... to disable sample mode

Description: The *HSCMP_SET_SAMPLE* ioctl command enables or disables sample mode at the high speed comparator module.

This command writes directly into the Sample Enable (SE) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_SAMPLE* command is implemented as a macro.

Example 5-717. HSCMP_SET_SAMPLE

```
ioctl(HSCMP_0, HSCMP_SET_SAMPLE, HSCMP_ENABLE);
```

This code enables sample mode at the HSCMP_2 module.

5.23.3.7 HSCMP_SET_WINDOWING - enable/disable windowing mode

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_WINDOWING,
           UWord16 param);
```

Arguments:

Table 5-819. HSCMP_SET_WINDOWING ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable windowing mode HSCMP_DISABLE ... to disable windowing mode

Description: The *HSCMP_SET_WINDOWING* ioctl command enables or disables windowing mode at the high speed comparator module.

This command writes directly into the Windowing Enable (WE) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_WINDOWING* command is implemented as a macro.

Example 5-718. HSCMP_SET_WINDOWING

```
ioctl(HSCMP_0, HSCMP_SET_WINDOWING, HSCMP_ENABLE);
```

This code enables windowing mode at the HSCMP_0 module.

5.23.3.8 HSCMP_SET_HIGH_SPEED - Sets power mode

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_HIGH_SPEED,
           UWord16 param);
```

Arguments:

Table 5-820. HSCMP_SET_HIGH_SPEED ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to select High speed mode HSCMP_DISABLE ... to select power savings mode

Description: The *HSCMP_SET_HIGH_SPEED* ioctl command sets power mode at the high speed comparator module. Parameter HSCMP_ENABLE sets the high speed comparator to high speed comparison mode and parameter HSCMP_DISABLE sets the high speed comparator to power saving mode.

This command writes directly into the Power Mode Select (PMODE) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_HIGH_SPEED* command is implemented as a macro.

Example 5-719. HSCMP_SET_HIGH_SPEED

```
ioctl(HSCMP_2, HSCMP_SET_HIGH_SPEED, HSCMP_ENABLE);
```

This code sets high speed comparision mode at the HSCMP_2 module.

5.23.3.9 HSCMP_SET_OUTPUT_PIN - Sets output pin source

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_OUTPUT_PIN,
           UWord16 param);
```

Arguments:

Table 5-821. HSCMP_SET_OUTPUT_PIN ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_COUNT ... to select count signal as output HSCMP_COUNTA ... to select counta signal as output

Description: The *HSCMP_SET_OUTPUT_PIN* ioctl command sets output source at the high speed comparator module. Parameter HSCMP_COUNT sets CMPO to equal COUNT (filtered comparator output) and parameter HSCMP_COUNTA sets CMPO to equal COUNTA (unfiltered comparator output).

This command writes directly into the Comparator Output Select (COS) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_OUTPUT_PIN* command is implemented as a macro.

Example 5-720. HSCMP_SET_OUTPUT_PIN

```
ioctl(HSCMP_1, HSCMP_SET_OUTPUT_PIN, HSCMP_COUNTA);
```

This code sets comparator output to equal COUNTA - unfiltered comparator output at the HSCMP_1 module.

5.23.3.10 HSCMP_SET_OUTPUT_ACTIVE - enabled/disable comparator output

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_SET_OUTPUT_ACTIVE,
           UWord16 param);
```

Arguments:

Table 5-822. HSCMP_SET_OUTPUT_ACTIVE ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable comparator output HSCMP_DISABLE ... to disable comparator output

Description: The *HSCMP_SET_OUTPUT_ACTIVE* ioctl command enables or disables comparator output at the high speed comparator module.

This command writes directly into the Comparator Output Pin Enable (OPE) bit of the HSCMP Control Register 1.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_SET_OUTPUT_ACTIVE* command is implemented as a macro.

Example 5-721. HSCMP_SET_OUTPUT_ACTIVE

```
ioctl(HSCMP_2, HSCMP_SET_OUTPUT_ACTIVE, HSCMP_ENABLE);
```

This code enables comparator output at the HSCMP_2 module.

5.23.3.11 HSCMP_INT_RISING_EDGE - enable/disable comparator rising edge interrupt

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_INT_RISING_EDGE,
     UWord16 param);
```

Arguments:

Table 5-823. HSCMP_INT_RISING_EDGE ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable rising edge interrupt HSCMP_DISABLE ... to disable rising edge interrupt

Description: The *HSCMP_INT_RISING_EDGE* ioctl command enables or disables comparator rising edge interrupt. This command uses periphSafeBitSet and periphSafeBitClear macros to prevent undesired clear interrut flags.

This command writes into the Comparator Interrupt Enable Rising (IER) bits of the HSCMP Status and Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_INT_RISING_EDGE* command is implemented as a macro.

Example 5-722. HSCMP_INT_RISING_EDGE

```
ioctl(HSCMP_1, HSCMP_INT_RISING_EDGE, HSCMP_ENABLE)
```

This code enables the rising-edge event interrupt at the HSCMP_1 module. The interrupt is generated any time the comparator output goes from low to high state.

5.23.3.12 HSCMP_INT_FALLING_EDGE - enable/disable comparator falling edge interrupt

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_INT_FALLING_EDGE,
     UWord16 param);
```

Arguments:

Table 5-824. HSCMP_INT_FALLING_EDGE ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_ENABLE ... to enable falling edge interrupt HSCMP_DISABLE ... to disable falling edge interrupt

Description: The *HSCMP_INT_FALLING_EDGE* ioctl command enables or disables comparator falling edge interrupt. This command uses periphSafeBitSet and periphSafeBitClear macros to prevent undesired clear interrut flags.

This command writes into the Comparator Interrupt Enable Falling (IEF) bits of the HSCMP Status and Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_INT_FALLING_EDGE* command is implemented as a macro.

Example 5-723. HSCMP_INT_FALLING_EDGE

```
ioctl(HSCMP_0, HSCMP_INT_FALLING_EDGE, HSCMP_DISABLE)
```

This code disables the falling-edge event interrupt at the HSCMP_0 module.

5.23.3.13 HSCMP_TEST_INT_FLAGS - read interrupt pending flags

Call(s):

```
UWord16 ioctl(const int *pModuleBase, HSCMP_TEST_INT_FLAGS,
          UWord16 param);
```

Arguments:

Table 5-825. HSCMP_TEST_INT_FLAGS ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use combination of the predefined constants: HSCMP_FLAG_RISING_EDGE |... to test rising edge interrupt flag HSCMP_FLAG_FALLING_EDGE ... to test falling edge interrupt flag

Description: The *HSCMP_TEST_INT_FLAGS* ioctl command returns the value of the selected comparator interrupt flags. This command returns state of the rising edge interrupt flag, when HSCMP_FLAG_RISING_EDGE is used as parameter. With the HSCMP_FLAG_FALLING_EDGE parameter value this command returns the value of falling edge interrupt flag.

This command returns the CFR or CFF bits of the HSCMP Status and Control Register.

Returns: UWord16 value of the selected interrupt status bits.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_TEST_INT_FLAGS* command is implemented as a macro.

Example 5-724. HSCMP_TEST_INT_FLAGS

```
if(ioctl(HSCMP_0, HSCMP_TEST_INT_FLAGS, HSCMP_FLAG_RISING_EDGE))
{
      ...
}
```

This code tests the rising-edge interrupt flag at the HSCMP_0 comparator module.

5.23.3.14 HSCMP_CLEAR_INT_FLAGS - clear interrupt flags

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_CLEAR_INT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-826. HSCMP_CLEAR_INT_FLAGS ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use combination of the predefined constants: HSCMP_FLAG_RISING_EDGE |... to clear rising edge interrupt flag HSCMP_FLAG_FALLING_EDGE ... to clearfalling edge interrupt flag

Description: The *HSCMP_CLEAR_INT_FLAGS* ioctl command clears the selected interrupt flags. This command clears the rising edge interrupt flag, when HSCMP_FLAG_RISING_EDGE is used as parameter. With HSCMP_FLAG_FALLING_EDGE parameter value this commaned clears falling edge interrupt flag.

This command writes the parameter value into the CRR and CFF bits of the HSCMP Status and Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_CLEAR_INT_FLAGS* command is implemented as a macro.

Example 5-725. HSCMP_CLEAR_INT_FLAGS

```
if(ioctl(HSCMP_0, HSCMP_TEST_INT_FLAGS, HSCMP_FLAG_RISING_EDGE))
{
     ...
}

ioctl(HSCMP_0, HSCMP_CLEAR_INT_FLAGS, HSCMP_FLAG_RISING_EDGE);
```

This code shows how pending interrupts are read, processed and cleared.

5.23.3.15 HSCMP_READ_OUTPUT - read immediate state of comparator output

Call(s):

```
UWord16 ioctl(const int *pModuleBase, HSCMP_READ_OUTPUT, NULL);
```

Arguments:

Table 5-827. HSCMP_READ_OUTPUT ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.

Description: The *HSCMP_READ_OUTPUT* ioctl command returns the immediate state of comparator output as it is read from the COUT bit of the HSCMP Status and Control Register. The returned value can only be tested in zero/non-zero conditions.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_READ_OUTPUT* command is implemented as a macro.

Example 5-726. HSCMP_READ_OUTPUT

```
if(ioctl(HSCMP_0, HSCMP_READ_OUTPUT, NULL))
{
      // output is high...
}
else
{
      // output is low...
}
```

This code shows how the immediate state of the comparator output may be tested.

5.23.3.16 HSCMP_WRITE_FILT_COUNTER - write Output Filter Counter

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_WRITE_FILT_COUNTER,
     UWord16 param);
```

Arguments:

Table 5-828. HSCMP_WRITE_FILT_COUNTER ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	Use one of the predefined constants: HSCMP_FILT_COUNTER_0 / HSCMP_FILT_COUNTER_1 / HSCMP_FILT_COUNTER_2 / HSCMP_FILT_COUNTER_3 / HSCMP_FILT_COUNTER_4 / HSCMP_FILT_COUNTER_5 / HSCMP_FILT_COUNTER_6 / HSCMP_FILT_COUNTER_7

Description: The *HSCMP_WRITE_FILT_COUNTER* ioctl command sets the filter sample count. Parameter HSCMP_FILT_COUNTER_0 disables the output filter. This command with parameter HSCMP_FILT_COUNTER1_1 sets the output filter to simple sampling mode - one consecutive samples must agree.

This command writes the parameter value into the FILT_CNT bits of the HSCMP Control Register 0.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_WRITE_FILT_COUNTER* ioctl command is implemented as a macro.

Example 5-727. HSCMP_WRITE_FILT_COUNTER

```
ioctl(HSCMP_0, HSCMP_WRITE_FILT_COUNTER, HSCMP_FILT_COUNTER_0);
```

This code disables output sample filter.

5.23.3.17 HSCMP_READ_FILT_COUNTER - read Output Filter Counter

Call(s):

```
UWord16 ioctl(const int *pModuleBase, HSCMP_READ_FILT_COUNTER,
              NULL);
```

Arguments:

Table 5-829. HSCMP_READ_FILT_COUNTER ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.

Description: The *HSCMP_READ_FILT_COUNTER* ioctl command reads and returns the value of the Output Filter counter.

This command reads and returns the FILT_CNT bits of the HSCMP Control Register.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_READ_FILT_COUNTER* ioctl command is implemented as a macro.

Example 5-728. HSCMP_READ_FILT_COUNTER

```
UWord16 filt = ioctl(HSCMP_0, HSCMP_READ_FILT_COUNTER, NULL);
ioctl(HSCMP_1, HSCMP_WRITE_FILT_COUNTER, filt);
```

This code copies the output filter counter value from the HSCMP_0 to HSCMP_1 module.

5.23.3.18 HSCMP_WRITE_FILT_REG - write Output Filter register

Call(s):

```
void ioctl(const int *pModuleBase, HSCMP_WRITE_FILT_REG,
     UWord16 param);
```

Arguments:

Table 5-830. HSCMP_WRITE_FILT_REG ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.
param	in	UWord16 register value.

Description: The *HSCMP_WRITE_FILT_REG* ioctl command writes directly to the Filter Period Register. The comparator output signal can be filtered by applying a digital “counting” filter. The FPR register contains the filter sample period value so this command may be used to reconfigure filtering feature in run-time. However, the output filter is typically configured statically (e.g. by using the Graphical Configuration Tool).

This command writes the parameter value directly to the Filter Period Register (FPR).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_WRITE_FILT_REG* ioctl command is implemented as a macro.

Example 5-729. HSCMP_WRITE_FILT_REG

```
ioctl(HSCMP_0, HSCMP_WRITE_FILT_REG, 0x0001);
```

This code writes value 0x01.directly to the Filter Period Register.

5.23.3.19 HSCMP_READ_FILT_REG - read Output Filter register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, HSCMP_READ_FILT_REG, NULL);
```

Arguments:

Table 5-831. HSCMP_READ_FILT_REG ioctl call arguments

*pModuleBase	in	The HSCMP module identifier. Use HSCMP_0, HCMP_1 or HSCMP_2.

Description: The *HSCMP_READ_FILT_REG* ioctl command reads and returns the value of the Output Filter Register.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *HSCMP_READ_FILT_REG* ioctl command is implemented as a macro.

Example 5-730. HSCMP_READ_FILT_REG

```
UWord16 filt = ioctl(HSCMP_0, HSCMP_READ_FILT_REG, NULL);
ioctl(HSCMP_1, HSCMP_WRITE_FILT_REG, filt);
```

This code copies the output filter settings from the HSCMP_0 to HSCMP_1 module.

5.23.4 HSCMP Driver Applications

There is one sample application demonstrating how the HSCMP module can be used in a user application. The supported DEMO boards for the HSCMP sample application are MC56F8006DEMO.

5.23.4.1 hscmp_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8006DEMO\hscmp_demo

This application shows how PWM generating a sawtooth signals. Analog Comparator module (HSCMP_0) is then used to compare the PWM outputs signals.

There are 6 LED diodes on this board. The led diodes PWM0 and PWM1 indicate values of PWM outputs.

LED diode PWM5 indicates that the main loop is working correctly.

LED diode PWM4 indicates that the PWM module is reloading correctly

LED diode PWM3 indicates that the HSCMP module interrupt is occuring periodicly.

5.24 DAC Driver

This section describes the API of MC56F802x/3x Digital-to-Analog Converter (DAC) on-chip module. The functionality of the DAC module itself is described in the **56F802x/3x Peripheral Reference Manual**.

5.24.1 Introduction

The MC56F802x/3x devices have one or two DAC modules which can be used internally in the device to set the Analog Comparator (CMP) threshold or a calibration voltage for Analog-to-Digital Converter (ADC). On some devices (e.g. MC56F8037), the signals for DACs are also available externally as a peripheral function of certain GPIO pins.

This section describes the DAC driver software, providing the low level software layer interfacing hardware with software.

5.24.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.24.3.

Table 5-832. DAC Module Base Address

Module base address of / for	MC56F801x	MC56F8023	MC56F802x/3x	MC56F83xx
DAC_0 (DAC0_BASE)	N/A	0xF1C0	0xF1C0	N/A
DAC_1 (DAC1_BASE)	N/A	N/A	0xF1D0	N/A

5.24.2.1 API Definition

The following header files are needed in order to use the DAC device driver:

Required Header File(s):

```
#include "qs.h"
#include "dac.h"
```

The following information may be found in the header file *dac.h*.

Public Data Structure(s):

None

5.24.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static DAC_0 and DAC_1 module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-833. DAC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
DAC_x_CTRL_INIT	UWord16	The initial value of the DAC Control Register
DAC_x_DATA_INIT	UWord16	Initial value of the DAC Data Register
DAC_x_STEP_INIT	UWord16	Initial value of the DAC Step Register
DAC_x_MINVAL_INIT	UWord16	Initial value of the DAC Minimum Value Register
DAC_x_MAXVAL_INIT	UWord16	Initial value of the DAC Maximum Value Register

Note - The "x" specifies the DAC module. This may be either 0 or 1. Note that the DAC_1 module may not be available on all devices.

5.24.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the DAC device modes or accesses the DAC register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-834. DAC Driver Arguments - ioctl

pModuleBase	in	DAC module identifier. Use DAC_0 or DAC_1.
cmd	in	Commands found in *dac.h* which are used to modify the DAC module status and control registers. See Table 5-835.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-835. ioctl commands

Cmd	param	Return	Description
DAC_INIT	NULL	None	Initializes DAC module by data from configuration file (appconfig.h).
DAC_MODULE	DAC_ENABLE/ DAC_DISABLE	None	Enables or disables the DAC device.
DAC_SET_DATA_FORMAT	DAC_RIGHT_ALIGNED_FMT / DAC_LEFT_ALIGNED_FMT	None	Selects how 12-bit DAC value is aligned in the DATA, STEP, MINVAL and MAXVAL registers.
DAC_SET_SYNC_SOURCE	DAC_IPBUS_SOURCE / DAC_SYNCIN_SOURCE	None	Selects the source of the DAC SYNC signal, which triggers an update of analog output.
DAC_SET_AUTO_MODE	DAC_AUTO_OFF / DAC_AUTO_SAWTOOTH1 / DAC_AUTO_SAWTOOTH2 / DAC_AUTO_TRIANGLE	None	Enables or disables an automatic DAC signal generation. When enabled, one of three signal shapes can be selected.
DAC_ENABLE_FILTER	UWord16 value 1-7 *DAC delay in IP-bus clocks Value of 0 results delay of 7 clocks.*	None	Enables a glitch-suppression filter. The parameter specifies how many clocks the DAC output is held unchanged after a new DATA value is written.
DAC_DISABLE_FILTER	NULL	None	Turns off the glitch-suppression filter.
DAC_WRITE_CONTROL_REG	UWord16	None	Directly writes to the DAC Control Register.

Table 5-835. ioctl commands (Continued)

Cmd	param	Return	Description
DAC_READ_CONTROL_REG	NULL	UWord16	Returns the value of the DAC Control Register.
DAC_WRITE_DATA	UWord16	None	Directly writes to the DAC Data Register.
DAC_READ_DATA	NULL	UWord16	Returns the value of the DAC Data Register.
DAC_WRITE_STEP	UWord16	None	Directly writes to the DAC Step Register.
DAC_READ_STEP	NULL	UWord16	Returns the value of the DAC Step Register.
DAC_WRITE_MINVAL	UWord16	None	Directly writes to the DAC Minimum Value Register.
DAC_READ_MINVAL	NULL	UWord16	Returns the value of the DAC Minimum Value Register.
DAC_WRITE_MAXVAL	UWord16	None	Directly writes to the DAC Maximum Value Register.
DAC_READ_MAXVAL	NULL	UWord16	Returns the value of the DAC Maximum Value Register.

5.24.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.24.3.1 DAC_INIT - initialize digital-to-analog converter module

Call(s):

```
void ioctl(const int *pModuleBase, DAC_INIT, NULL);
```

Arguments:

Table 5-836. DAC_INIT ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_INIT* ioctl command executes the initialization function, which initializes all configured DAC registers by the values defined in *appconfig.h*. It is intended for static configuration of the DAC module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_INIT* ioctl command is implemented as a function call. Only necessary DAC registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-731. DAC_INIT

```
ioctl(DAC_0, DAC_INIT, NULL);
ioctl(DAC_1, DAC_INIT, NULL);
```

This code initializes the DAC modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.24.3.2 DAC_MODULE - enable/disable DAC device

Call(s):

```
void ioctl(const int *pModuleBase, DAC_MODULE, UWord16 param);
```

Arguments:

Table 5-837. DAC_MODULE ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	Use DAC_ENABLE to enable or DAC_DISABLE to disable DAC module.

Description: The *DAC_MODULE* ioctl command enables (DAC_ENABLE) or disables (DAC_DISABLE) the DAC module. Before the DAC module is used, the DAC device should be configured (in disabled state) and then enabled. Also, the DAC module can be enabled already during the initialization procedure.

This command writes directly into the Power-Down (PDN) bit of the DAC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_MODULE* command is implemented as a macro.

Example 5-732. DAC_MODULE

```
/* configure desired DAC parameters */
/* ... */
/* enable it */
ioctl(DAC_0, DAC_MODULE, DAC_ENABLE);
```

This code enables the DAC_0 module. Before enabling, the DAC configuration should be taken.

5.24.3.3 DAC_SET_DATA_FORMAT - select 12-bit data alignment in 16-bit word

Call(s):

```
void ioctl(const int *pModuleBase, DAC_SET_DATA_FORMAT,
     UWord16 param);
```

Arguments:

Table 5-838. DAC_SET_DATA_FORMAT ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	Use one of the predefined constants: DAC_RIGHT_ALIGNED_FMT ... use bits 11:0 DAC_LEFT_ALIGNED_FMT ... use bits 15:4

Description: The *DAC_SET_DATA_FORMAT* ioctl command configures the data alignment for the DATA, STEP, MINVAL and MAXVAL Registers. The alignment defines what bits of the registers contain the 12-bit DAC value.

In the right-aligned mode, the 12-bit value is located on bits 11:0 within each register. This default mode is suitable for integer arithmetic when custom scaling methods are used.

In the left-aligned mode, the 12-bit value is located on bits 15:4 within each register. This mode is suitable for fractional values and results.

This command writes directly into the Data-Format (FORMAT) bit of the DAC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_SET_DATA_FORMAT* command is implemented as a macro.

Example 5-733. DAC_SET_DATA_FORMAT

```
ioctl(DAC_0, DAC_SET_DATA_FORMAT, DAC_RIGHT_ALIGNED_FMT);
...

ioctl(DAC_0, DAC_WRITE_DATA, 0xfff);
```

This applies maximum analog value on the DAC_0 output. The value is specified in the right-aligned format.

5.24.3.4 DAC_SET_SYNC_SOURCE - select DAC trigger signal source

Call(s):

```
void ioctl(const int *pModuleBase, DAC_SET_SYNC_SOURCE,
     UWord16 param);
```

Arguments:

Table 5-839. DAC_SET_SYNC_SOURCE ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	Use one of the predefined constants: DAC_IPBUS_SOURCE ... IP Bus clock is used as DAC trigger signal DAC_SYNCIN_SOURCE ... SYNC_IN is used as DAC trigger signal

Description: The *DAC_SET_SYNC_SOURCE* ioctl command enables or disables the internal SYNC_IN signal to be used to trigger an update of the buffered data to the analog DAC output. When disabled (DAC_IPBUS_SOURCE), the IP bus clock is used as a DAC trigger, so the values written to DATA register is presented to DAC on next clock cycle. When enabled (DAC_SYNCIN_SOURCE), the rising edge of SYNC_IN signal updates the DAC output.

The SYNC_IN signal source may be further configured in the SYS module of MC56F802x/3x to be a PIT rollover event, PWM reload event or Timer A output.

Note that an additional glitch-suppression filter may be applied and may cause a latency between the trigger signal and actual update of the DAC output.

This command writes directly into the Synchronization-Enable (SYNC_EN) bit of the DAC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_SET_SYNC_SOURCE* command is implemented as a macro.

Example 5-734. DAC_SET_SYNC_SOURCE

```
ioctl(SYS, SYS_SET_DAC0SYNC_SOURCE, SYS_DAC0SYNCSRC_PWMSYNC);
ioctl(DAC_0, DAC_SET_SYNC_SOURCE, DAC_SYNCIN_SOURCE);
```

This example configures the DAC_0 trigger signal to be a PWM synchronization pulse.

5.24.3.5 DAC_SET_AUTO_MODE - select automatic operation mode of the DAC module

Call(s):

```
void ioctl(const int *pModuleBase, DAC_SET_AUTO_MODE,
     UWord16 param);
```

Arguments:

Table 5-840. DAC_SET_AUTO_MODE ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	Use one of the predefined constants: DAC_AUTO_OFF ... manual write to DATA Register updates DAC DAC_AUTO_SAWTOOTH1 ... sawtooth signal /|/|/| generated DAC_AUTO_SAWTOOTH2 ... sawtooth signal |\/|\/| generated DAC_AUTO_TRIANGLE ... triangle signal /\/\/\ generated

Description: The *DAC_SET_AUTO_MODE* ioctl command may be used to enable or disable generating a shaped analog signal on the DAC output. When disabled (DAC_AUTO_OFF), the user application is responsible to update the DATA Register and to control the DAC analog output this way. When enabled, one of three signal shapes can be selected.

There are three registers which define actual values for an automatically generated signal. The DATA is automatically incremented or decremented (or both) always between MINVAL and MAXVAL limits, with the step defined by the STEP Register. The DAC_AUTO_TRIANGLE selects both incrementing and decrementing mode, so a “triangle” signal is generated. The two DAC_AUTO_SAWTOOTH modes use STEP in only one direction and an immediate jump-change of the DATA value in the other direction. See the **56F802x/3x Peripheral Reference Manual** for more details.

This command writes directly into the UP, DOWN and AUTO bits of the DAC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_SET_AUTO_MODE* command is implemented as a macro.

Example 5-735. DAC_SET_AUTO_MODE

```
ioctl(SYS, SYS_SET_DAC0SYNC_SOURCE, SYS_DAC0SYNCSRC_PIT0);
ioctl(DAC_0, DAC_SET_SYNC_SOURCE, DAC_SYNCIN_SOURCE);

ioctl(DAC_0, DAC_WRITE_MINVAL, 1000);
ioctl(DAC_0, DAC_WRITE_MAXVAL, 4000);
ioctl(DAC_0, DAC_WRITE_STEP, 10);

ioctl(DAC_0, DAC_SET_AUTO_MODE, DAC_AUTO_TRIANGLE);
```

This example configures the DAC_0 to generate triangle output signal between voltage levels corresponding to DATA values 1000 and 4000. Each step (value 10) is triggered by the PIT rollover event.

5.24.3.6 DAC_ENABLE_FILTER - enable and configure the glitch-suppression filter

Call(s):

```
void ioctl(const int *pModuleBase, DAC_ENABLE_FILTER,
     UWord16 param);
```

Arguments:

Table 5-841. DAC_ENABLE_FILTER ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	Use Uword16 value in range 1-7. If a 0 is used, it will be converted to a default value of 7.

Description: The DAC_ENABLE_FILTER ioctl command configures and enables the glitch-suppression filter on the DAC output. The glitch suppression filter introduces a latency between the DATA Register update (both manual or automatic) and the actual DAC output update. The parameter value (1-7) represents a number of IPBus clock cycles the DAC output is held unchanged after new data is presented. Approximately 240 nsec is required for worst case settling of the DAC output; therefore, a value of seven should be used for a default 32 MHz operation.

Note: When using the glitch filter, be sure the filter count is less than the update count (SYNC_IN period) otherwise the DAC output will not be updated at all. See the **56F802x/3x Peripheral Reference Manual** for more details.

This command writes the parameter value directly into the FILT_CNT bit-field and sets the FILT_EN bit in the DAC Control Register.

Returns: None.

Range Issues: Use integer value 1-7. Value of zero will be converted to 7.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_ENABLE_FILTER* command is implemented as a macro.

Example 5-736. DAC_ENABLE_FILTER

```
ioctl(DAC_0, DAC_ENABLE_FILTER, 7);
```

This example enables the glitch-suppression filter and sets the DAC_0 output latency to 7 IP bus clocks.

5.24.3.7 DAC_DISABLE_FILTER - disable the glitch-suppression filter

Call(s):

```
void ioctl(const int *pModuleBase, DAC_DISABLE_FILTER, NULL);
```

Arguments:

Table 5-842. DAC_DISABLE_FILTER ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The DAC_DISABLE_FILTER ioctl command disables the glitch-suppression filter. See more details about the filter at DAC_ENABLE_FILTER ioctl command description or in the **56F802x/3x Peripheral Reference Manual**.

This command clears the FILT_EN bit in the DAC Control Register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The *DAC_DISABLE_FILTER* command is implemented as a macro.

Example 5-737. DAC_DISABLE_FILTER

```
ioctl(DAC_0, DAC_DISABLE_FILTER, NULL);
```

This example disables the glitch-suppression filter of the DAC_0 module.

5.24.3.8 DAC_WRITE_CONTROL_REG - write DAC Control Register

Call(s):

```
void ioctl(const int *pModuleBase, DAC_WRITE_CONTROL_REG,
           UWord16 param);
```

Arguments:

Table 5-843. DAC_WRITE_CONTROL_REG ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	UWord16

Description: The *DAC_WRITE_CONTROL_REG* ioctl command writes the parameter value into the DAC Control Register.

Returns: None

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_WRITE_CONTROL_REG command is implemented as a macro.

Example 5-738. DAC_WRITE_CONTROL_REG

```
ioctl(DAC_0, DAC_WRITE_CONTROL_REG, 0xF01C);
```

This code writes 0xF01C to the DAC Control Register.

5.24.3.9 DAC_READ_CONTROL_REG - read DAC Control Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DAC_READ_CONTROL_REG,
      NULL);
```

Arguments:

Table 5-844. DAC_READ_CONTROL_REG ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_READ_CONTROL_REG* ioctl command reads and returns the value of the DAC Control Register.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_READ_CONTROL_REG command is implemented as a macro.

Example 5-739. DAC_READ_CONTROL_REG

```
UWord16 reg;

reg = ioctl(DAC_0, DAC_READ_CONTROL_REG, NULL);
ioctl(DAC_1, DAC_WRITE_CONTROL_REG, reg);
```

This code copies the DAC Control Register content from DAC_0 to DAC_1.

5.24.3.10 DAC_WRITE_DATA - write DAC DATA Register

Call(s):

```
void ioctl(const int *pModuleBase, DAC_WRITE_DATA,
           UWord16 param);
```

Arguments:

Table 5-845. DAC_WRITE_DATA ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	UWord16

Description: The *DAC_WRITE_DATA* ioctl command writes the parameter value into the DAC DATA Register. This causes the DAC output to be changed after the next update-trigger signal which is either SYNC_IN or IP Bus clock. See description of the DAC_SET_SYNC_SOURCE and DAC_SET_AUTO_MODE commands for more details.

Returns: None

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT command, the 12-bit data being written by the DAC_WRITE_DATA command should be left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_WRITE_DATA command is implemented as a macro.

Example 5-740. DAC_WRITE_DATA

```
ioctl(DAC_0, DAC_SET_DATA_FORMAT, DAC_RIGHT_ALIGNED_FMT);
ioctl(DAC_0, DAC_WRITE_DATA, 0x0800);
```

This code writes 0x800 into the DAC_0 DATA Register which causes the voltage output to be set in the middle of applicable output range.

5.24.3.11 DAC_READ_DATA - read DAC DATA Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DAC_READ_DATA, NULL);
```

Arguments:

Table 5-846. DAC_READ_DATA ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_READ_DATA* ioctl command reads and returns a value of the DAC Data Register.

Returns: The register value as UWord16.

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit data being returned by the DAC_READ_DATA command is left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_READ_DATA command is implemented as a macro.

Example 5-741. DAC_READ_DATA

```
UWord16 data = ioctl(DAC_0, DAC_READ_DATA, NULL);
```

This code reads the DAC_0 DATA Register.

5.24.3.12 DAC_WRITE_STEP - write DAC STEP Register

Call(s):

```
void ioctl(const int *pModuleBase, DAC_WRITE_STEP,
           UWord16 param);
```

Arguments:

Table 5-847. DAC_WRITE_STEP ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	UWord16

Description: The *DAC_WRITE_STEP* ioctl command writes the parameter value into the DAC STEP Register. This value is used anytime the DAC is switched to an automatic mode (see the DAC_SET_AUTO_MODE ioctl command) as an auto increment or decrement value.

Returns: None

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit value being written by DAC_WRITE_STEP command should be left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_WRITE_STEP command is implemented as a macro.

Example 5-742. DAC_WRITE_STEP

```
ioctl(DAC_0, DAC_SET_DATA_FORMAT, DAC_RIGHT_ALIGNED_FMT);
ioctl(DAC_0, DAC_WRITE_STEP, 1);
```

This code writes 1 into the DAC_0 STEP Register.

5.24.3.13 DAC_READ_STEP - read DAC STEP Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DAC_READ_STEP, NULL);
```

Arguments:

Table 5-848. DAC_READ_STEP ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_READ_STEP* ioctl command reads and returns a value of the DAC Step Register.

Returns: The register value as UWord16

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT command, the 12-bit value being returned by the DAC_READ_STEP command is left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_READ_STEP command is implemented as a macro.

Example 5-743. DAC_READ_STEP

```
UWord16 step = ioctl(DAC_0, DAC_READ_STEP, NULL);
```

This code reads the DAC_0 STEP Register.

5.24.3.14 DAC_WRITE_MINVAL - write DAC MINVAL Register

Call(s):

```
void ioctl(const int *pModuleBase, DAC_WRITE_MINVAL,
           UWord16 param);
```

Arguments:

Table 5-849. DAC_WRITE_MINVAL ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	UWord16

Description: The *DAC_WRITE_MINVAL* ioctl command writes the parameter value into the DAC Minimum Value Register. This value is used anytime the DAC is switched to an automatic mode (see the DAC_SET_AUTO_MODE ioctl command) as the bottom limit value.

Returns: None

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit value being written by the DAC_WRITE_MINVAL command should be left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_WRITE_MINVAL command is implemented as a macro.

Example 5-744. DAC_WRITE_MINVAL

```
ioctl(DAC_0, DAC_SET_DATA_FORMAT, DAC_RIGHT_ALIGNED_FMT);
ioctl(DAC_0, DAC_WRITE_MINVAL, 1000);
```

This code writes 1000 into the DAC_0 MINVAL Register.

5.24.3.15 DAC_READ_MINVAL - read DAC MINVAL Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DAC_READ_MINVAL, NULL);
```

Arguments:

Table 5-850. DAC_READ_MINVAL ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_READ_MINVAL* ioctl command reads and returns a value of the DAC Minimum Value Register.

Returns: The register value as UWord16.

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit value being returned by the DAC_READ_MINVAL command is left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_READ_MINVAL command is implemented as a macro.

Example 5-745. DAC_READ_MINVAL

```
UWord16 minval = ioctl(DAC_0, DAC_READ_MINVAL, NULL);
```

This code reads the DAC_0 MINVAL Register.

5.24.3.16 DAC_WRITE_MAXVAL - write DAC MAXVAL Register

Call(s):

```
void ioctl(const int *pModuleBase, DAC_WRITE_MAXVAL,
           UWord16 param);
```

Arguments:

Table 5-851. DAC_WRITE_MAXVAL ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.
param	in	UWord16

Description: The *DAC_WRITE_MAXVAL* ioctl command writes the parameter value into the DAC Maximum Value Register. This value is used anytime the DAC is switched to an automatic mode (see the DAC_SET_AUTO_MODE ioctl command) as the top limit value.

Returns: None

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit value being written by the DAC_WRITE_MAXVAL command should be left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_WRITE_MAXVAL command is implemented as a macro.

Example 5-746. DAC_WRITE_MAXVAL

```
ioctl(DAC_0, DAC_SET_DATA_FORMAT, DAC_RIGHT_ALIGNED_FMT);
ioctl(DAC_0, DAC_WRITE_MAXVAL, 4000);
```

This code writes 4000 into the DAC_0 MAXVAL Register.

5.24.3.17 DAC_READ_MAXVAL - read DAC MAXVAL Register

Call(s):

```
UWord16 ioctl(const int *pModuleBase, DAC_READ_MAXVAL, NULL);
```

Arguments:

Table 5-852. DAC_READ_MAXVAL ioctl call arguments

*pModuleBase	in	The DAC module identifier. Use DAC_0 or DAC_1.

Description: The *DAC_READ_MAXVAL* ioctl command reads and returns a value of the DAC Maximum Value Register.

Returns: The register value as UWord16.

Range Issues: Depending on the data format selected by the DAC_SET_DATA_FORMAT ioctl command, the 12-bit value being returned by the DAC_READ_MAXVAL command is left- or right-aligned within the UWord16 value.

Special Issues: This command is applicable only on MC56F802x/3x.

Design/Implementation: The DAC_READ_MAXVAL command is implemented as a macro.

Example 5-747. DAC_READ_MAXVAL

```
UWord16 maxval = ioctl(DAC_0, DAC_READ_MAXVAL, NULL);
```

This code reads the DAC_0 MAXVAL Register.

5.24.4 DAC Driver Applications

There are three sample applications demonstrating how the DAC module can be used in a user application. The supported EVM boards for the DAC sample applications are MC56F8025DEMO and MC56F8037EVM.

5.24.4.1 dac_pit_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\dac_pit_demo

This application shows how DAC_0 and DAC_1 are driven by PIT_0 and PIT_1 timers in an automatic mode, generating a sawtooth signal.

5.24.4.2 dac_cmp_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\dac_cmp_demo

This application shows how DAC_0 and DAC_1 are driven by PIT_0 timer in an automatic mode, generating a sawtooth signal. Analog Comparator module (CMP) is then used to compare the DAC_0 and DAC_1 signals and sets one user LED accordingly.

5.24.4.3 adc_fmstr_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\EVM\adc_fmstr_demo

Again, this application shows how DAC_0 and DAC_1 are driven by PIT_0 counter in an automatic mode, generating a sawtooth signal. The ANA7 and ANB7 channels are sourced from DAC_0 and DAC_1 and the ADC-measured values may be displayed in the FreeMASTER visualization tool.

5.25 MSCAN Driver

This section describes the API of MC56F803x MSCAN on-chip module. The functionality of the MSCAN module itself is described in the **56F802x/3x Peripheral Reference Manual**.

5.25.1 Introduction

The MC56F803x devices have one MSCAN module which is a communication controller implementing the CAN protocol version 2.0 A/B. This module is slightly different (modernized) version when comparing it to an older implementation on DSP56F800 devices. The MSCAN module is also completely different module comparing it with the FlexCAN modules of the MC56F83xx family.

This section describes the MSCAN driver software, providing the low level software layer interfacing hardware with software.

5.25.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.25.3.

Table 5-853. MSCAN Module Base Address

Module base address of / for	MC56F801x	MC56F803x	MC56F83xx
MSCAN (CAN_BASE)	N/A	0xF800	N/A

5.25.2.1 MSCAN Bit-Timing

The MSCAN module supports a variety of means to setup the bit timing required by the CAN protocol. The main MSCAN clock is called serial clock or *sclock*. The ratio between system clock and *sclock* can be specified using the 6-bit Baud Rate Prescaler value in the CANBTR0 register.

sclock = system clock / (CANBTR0.BRP+1)

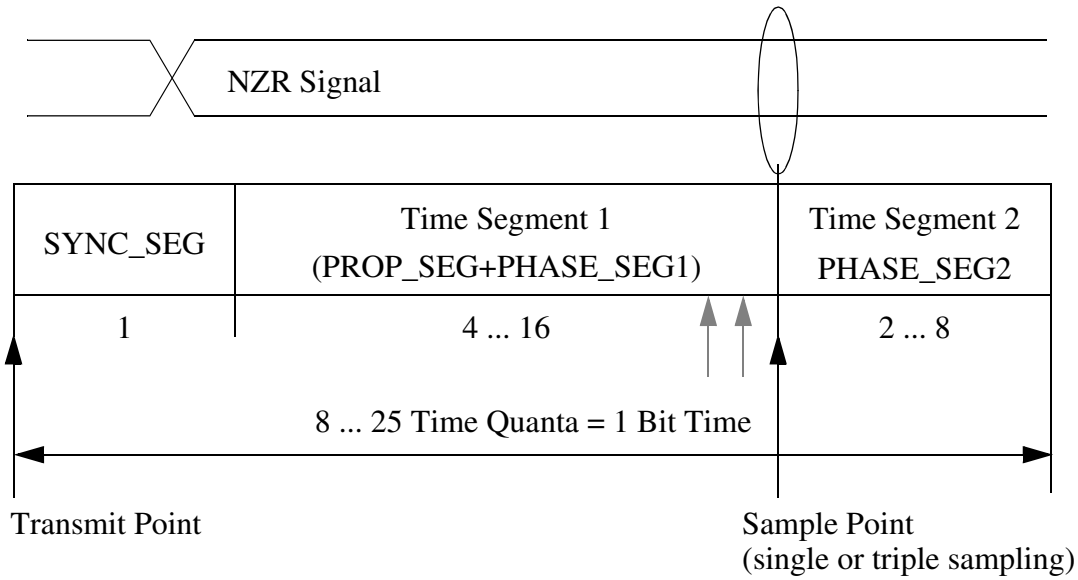
A single *sclock* cycle defines a basic time unit called “time quantum” (tq). All other MSCAN timing parameters are measured in the time quanta units.

1/sclock = 1 time quantum

The nominal bit rate of the CAN bus is based on the nominal bit time interval.

nominal bit rate = 1 / nominal bit time

The nominal bit time is split to four non-overlapping time intervals, each measured in time quanta units.



SYNC_SEG - Synchronization Segment

This part of the bit time is used to synchronize the various nodes on the bus. An edge is expected to lie within this segment. This part is always 1 tq long.

SYNC_SEG = 1 [tq]

PROP_SEG - Propagation Segment

This part of the bit time is used to compensate for the physical delay times within the network. It is twice the sum of the signal's propagation time on the bus line, the input comparator delay, and the output driver delay. The PROP_SEG is included in the PHASE_SEG1 in the MSCAN implementation.

PHASE_SEG1, PHASE_SEG2 - Phase Buffer Segments

These Phase Buffer Segments are used to compensate the edge phase errors. These segments can be lengthened or shortened by re-synchronization (SJW parameter). The length of the Phase Buffer Segments can be configured in the TSEG1 and TSEG2 fields of the CANBTR1 register.

PHASE_SEG1 + PROP_SEG = CANBTR1.TSEG1 + 1 [tq]

PHASE_SEG2 = CANBTR1.TSEG2 + 1 [tq]

RE-SYNCHRONIZATION JUMP WIDTH (SJW)

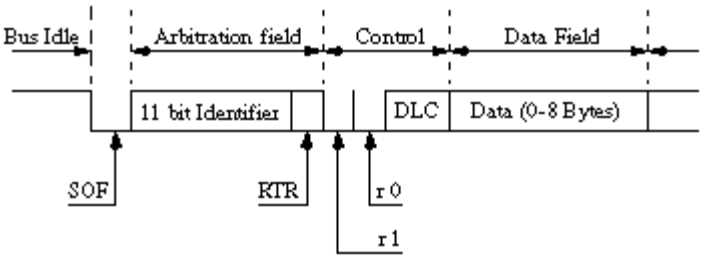
As a result of re-synchronization, PHASE_SEG1 may be lengthened or PHASE_SEG2 may be shortened. The amount of lengthening or shortening of the Phase Buffer Segments has an upper bound given by the Re-synchronization Jump Width. This parameter can be configured in the SJW field of the CANBTR0 register.

RJW = CANBTR0.SJW + 1 [tq]

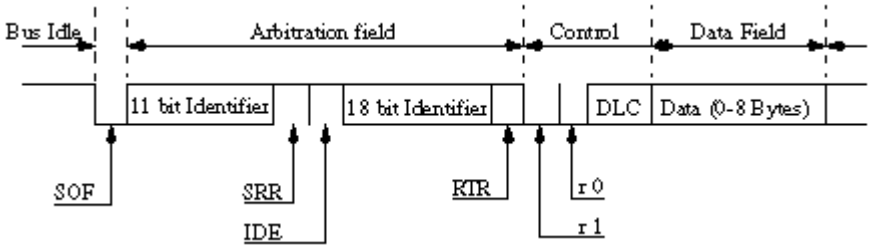
For more information about the bit timing requirements, see the CAN 2AB standard document or Freescale Application Note number AN1798 (order number AN1798/D).

5.25.2.2 CAN Message Identifiers

CAN Standard 2.0 A/B defines two kinds of CAN message identifiers. Part A of the standard defines “standard” 11 bit message identifier and part B defines “extended” 29 bit identifier. The CAN message header contains control bit named “ID Extended” (IDE) which, when set, identifies message with 29 bit extended identifier. In Part A-message format the IDE bit is reserved for future use (“r1”) and has always a value of zero.



The Remote Transmit Request (RTR) bit, which follows the 11 bit identifier in the standard ID messages is replaced with “dummy” Substitute Remote Request (SRR) bit in extended 29 bit ID messages. The SRR is always transmitted as one. For extended ID messages, 18 ID bits added and a new RTR bit are appended after IDE bit.



There are several MSCAN registers where the message ID is to be specified. The examples are the Receive Message Buffer ID Registers, Transmit Buffer ID Registers or ID Mask and Acceptance Registers. Various registers containing the message ID have always the same format, which reflects the physical bit sequence as captured-from or generated-onto the CAN bus. A care must be taken when a numerical ID is to be supplied to registers or if a register-contained value is to be understood as a numerical CAN message identifier. The MSCAN low-level driver as well as this document use the term “ID-raw” (IDR) whenever the register-contained value is referenced. The ID-raw is 32-bit value and consists of four bytes which must be treated differently when its IDE bit (bit 19) is 0 (standard ID) or 1 (extended ID).

	7	6	5	4	3	2	1	0
IDR0	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3

IDR1	ID2	ID1	ID0	RTR	IDE (=0)			
IDR2								
IDR3								

Table 5-854. ID-raw Registers with Standard CAN ID

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
IDR0	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
IDR1	ID20	ID19	ID18	SRR (=1)	IDE (=1)	ID17	ID16	ID15
IDR2	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
IDR3	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

Table 5-855. ID-raw Registers with Extended CAN ID

The MSCAN low-level driver of DSP56800E_Quick_Start contains two inline methods to convert numerical representation of the message ID from and to the 32-bit value, which is suitable for writing into the IDR0-3 registers.

```
inline UWord32 MSCAN_Id2Idr(UWord32 id);
```

This inline function converts the numerical representation of the ID value into the “raw” format suitable for the IDR registers. The most significant bit (bit31) of the ID value is used to identify extended message identifier. The valid range of the input parameter is 0x000...0x7ff (standard identifier) and 0x80000000...0x9fffffff (extended identifier). The symbol MSCAN_ID_EXT is defined in *mscan.h* as the MSB (0x80000000), so it can be OR-ed with the numerical ID representation when specifying extended identifiers.

```
inline UWord32 MSCAN_Idr2Id(UWord32 idr);
```

This inline function parses the “raw” value read from two MSCAN IDR registers into a numerical representation. The MSB of the returned value is set for extended identifiers.

The MSCAN ioctl commands accept and return the numerical representation of the ID values (plus the MSB bit to identify extended identifier) and use the conversion functions described above implicitly (e.g. *MSCANMB_SET_ID* command[1]). The ioctl commands operating directly with the ID-raw value without further conversions have the *_RAW* suffix (e.g. *MSCANMB_GET_ID_RAW*).

1. An exception is the MSCANMB_GET_ID command, which is implemented as a optimized assembly function in *mscan.c*.

5.25.2.3 API Definition

The following header files are needed in order to use the MSCAN device driver:

Required Header File(s):

```
#include "qs.h"
#include "mscan.h"
```

General-Use Macros and inline functions:

```
inline UWord32 MSCAN_Id2Idr(UWord32 id);
inline UWord32 MSCAN_Idr2Id(UWord32 idr);
inline UWord32 MSCAN_Id2Idr_V(UWord32 id);
inline UWord32 MSCAN_Idr2Id_V(UWord32 idr);
```

5.25.2.4 Configuration Items

This section summarizes the symbols used in macro definitions for the static MSCAN module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-856. MSCAN Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
MSCAN_CANCTL0_INIT	UWord16	The initial value of the MSCAN Control Register 0.
MSCAN_CANCTL1_INIT	UWord16	The initial value of the MSCAN Control Register 1.
MSCAN_CANBTR0_INIT	UWord16	The initial value of the MSCAN Bus Timing Register 0.
MSCAN_CANBTR1_INIT	UWord16	The initial value of the MSCAN Bus Timing Register 1.
MSCAN_CANIDAC_INIT	UWord16	The initial value of the MSCAN Identifier Acceptance Control Register.
MSCAN_CANRIER_INIT	UWord16	The initial value of the MSCAN Receiver Interrupt Enable Register.
MSCAN_CANTCR_INIT	UWord16	The initial value of the MSCAN Transmit Interrupt Enable Register.
MSCAN_CANIDAR0_INIT MSCAN_CANIDAR1_INIT MSCAN_CANIDAR2_INIT MSCAN_CANIDAR3_INIT MSCAN_CANIDAR4_INIT MSCAN_CANIDAR5_INIT MSCAN_CANIDAR6_INIT MSCAN_CANIDAR7_INIT	UWord16	The initial values of the MSCAN Identifier Acceptance Registers 0-7.

Table 5-856. MSCAN Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
MSCAN_CANIDMR0_INIT MSCAN_CANIDMR1_INIT MSCAN_CANIDMR2_INIT MSCAN_CANIDMR3_INIT MSCAN_CANIDMR4_INIT MSCAN_CANIDMR5_INIT MSCAN_CANIDMR6_INIT MSCAN_CANIDMR7_INIT	UWord16	The initial values of the MSCAN Identifier Mask Registers 0-7.

5.25.2.5 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the MSCAN device modes or accesses the MSCAN register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-857. MSCAN Driver Arguments - ioctl

pModuleBase	in	MSCAN module identifier. Use MSCAN.
cmd	in	Commands found in *mscan.h* which are used to modify the MSCAN module status and control registers. See Table 5-858.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-858. ioctl commands

Cmd	param	Return	Description
MSCAN_INIT	NULL	None	Initializes the MSCAN module by data from configuration file (app-config.h).
MSCAN_SOFT_RESET	MSCAN_ON / MSCAN_OFF	None	Enters or leaves the soft-reset state.
initialization-time-only commands (CAN must be in Reset state)			
MSCAN_DEVICE	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables the MSCAN peripheral module.
MSCAN_WAKEUP_FILTER	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables low-pass filter on wake-up condition (dominant pulse).
MSCAN_MANUAL_BOFF_RECOVERY	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables manual bus-off recovery mode.
MSCAN_LISTEN_ONLY_MODE	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables listen-only mode.
MSCAN_LOOPBACK_MODE	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables internal loop-back mode.
MSCAN_SET_CLOCK_SOURCE	MSCAN_IPBUS / MSCAN_XTAL	None	Selects clock source for the CAN module.
MSCAN_SET_PRESCALER	UWord16 in range 1...64	None	Sets bit-rate prescaler.
MSCAN_SET_SJW	UWord16 in range 1...4 or MSCAN_SJW_1...4	None	Sets synchronization jump width period.
MSCAN_SET_TSEG1	UWord16 in range 1...16 or MSCAN_TSEG_1...16	None	Sets TSEG1 segment.
MSCAN_SET_TSEG2	UWord16 in range 1...8 or MSCAN_TSEG_1...8	None	Sets TSEG2 segment.
MSCAN_SET_SAMPLING	MSCAN_1SAMP_PER_BIT / MSCAN_3SAMPS_PER_BIT	None	Sets number of bus sampling per one data bit.
MSCAN_SET_ACC_MODE	MSCAN_ACC_MODE_2X32 / MSCAN_ACC_MODE_4X16 / MSCAN_ACC_MODE_8X8 / MSCAN_ACC_MODE_CLOSED	None	Configures the CAN identifier acceptance filter.

Table 5-858. ioctl commands (Continued)

Cmd	param	Return	Description
MSCAN_SET_ACC_MASKR_32_0 MSCAN_SET_ACC_MASKR_32_1	UWord32	None	Sets the 32-bit Mask and ID acceptance filter values (filter 0 or 1) when filter is configured as MSCAN_ACC_MODE_2X32.
MSCAN_SET_ACC_IDR_32_0 MSCAN_SET_ACC_IDR_32_1			
MSCAN_SET_ACC_MASKR_16_0 MSCAN_SET_ACC_MASKR_16_1 MSCAN_SET_ACC_MASKR_16_2 MSCAN_SET_ACC_MASKR_16_3	UWord16	None	Sets the 16-bit Mask and ID acceptance filter values (filters 0..3) when filter is configured as MSCAN_ACC_MODE_4X16.
MSCAN_SET_ACC_IDR_16_0 MSCAN_SET_ACC_IDR_16_1 MSCAN_SET_ACC_IDR_16_2 MSCAN_SET_ACC_IDR_16_3			
MSCAN_SET_ACC_MASKR_8_0 MSCAN_SET_ACC_MASKR_8_1 MSCAN_SET_ACC_MASKR_8_2 MSCAN_SET_ACC_MASKR_8_3 MSCAN_SET_ACC_MASKR_8_4 MSCAN_SET_ACC_MASKR_8_5 MSCAN_SET_ACC_MASKR_8_6 MSCAN_SET_ACC_MASKR_8_7	UWord16 (eight LSB used)	None	Sets the 8-bit Mask and ID acceptance filter values (filters 0..7) when filter is configured as MSCAN_ACC_MODE_8X8.
MSCAN_SET_ACC_IDR_8_0 MSCAN_SET_ACC_IDR_8_1 MSCAN_SET_ACC_IDR_8_2 MSCAN_SET_ACC_IDR_8_3 MSCAN_SET_ACC_IDR_8_4 MSCAN_SET_ACC_IDR_8_5 MSCAN_SET_ACC_IDR_8_6 MSCAN_SET_ACC_IDR_8_7			
run-time commands			
MSCAN_SLEEP	MSCAN_ON / MSCAN_OFF	None	Enters or leaves the sleeping state.
MSCAN_GET_SLEEP_MODE	NULL	UWord16	Gets sleeping state. Returns one of the following constants: MSCAN_NOT_SLEEPING / MSCAN_SLEEPING / MSCAN_SLEEP_REQUESTED
MSCAN_AUTO_WAKEUP	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables automatic wake-up on bus activity.
MSCAN_TIMESTAMP_TIMER	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables internal timestamp timer (clocked by CAN bit-clock).
MSCAN_TEST_SYNCH	NULL	UWord16 zero/non-zero	Tests is the CAN module is synchronized to CAN bus.
MSCAN_TEST_RXACT	NULL	UWord16 zero/non-zero	Tests if the CAN module is just receiving a CAN message.

Table 5-858. ioctl commands (Continued)

Cmd	param	Return	Description
MSCAN_TEST_RXFRM	NULL	UWord16 zero/non-zero	Tests if the CAN message was received since the last clearing by MSCAN_CLEAR_RXFRM.
MSCAN_CLEAR_RXFRM	NULL	None	Clears frame-received flag (not an interrupt flag).
MSCAN_STOP_IN_WAIT	MSCAN_ENABLE / MSCAN_DISABLE	None	Enables or disables the CAN module shutdown in CPU wait mode.
MSCAN_ERINT_ENABLE MSCAN_ERINT_DISABLE	Any combination of: MSCAN_WAKEUP_INT | MSCAN_STATCHNG_INT | MSCAN_OVERRUN_INT | MSCAN_RXFULL_INT	None	Enables or disable the selected Error or Receive interrupts.
MSCAN_ERINT_SET_RSTATE_MODE MSCAN_ERINT_SET_TSTATE_MODE	One of: MSCAN_STAT_VOID / MSCAN_STAT_BUSOFF / MSCAN_STAT_ERR / MSCAN_STAT_ALL	None	Configures receiver and transmitter conditions for raising the CAN status-change interrupt (MSCAN_STATCHNG_INT).
MSCAN_TINT_ENABLE MSCAN_TINT_DISABLE	Any combination of: MSCAN_TXEMPTY0_INT | MSCAN_TXEMPTY1_INT | MSCAN_TXEMPTY2_INT | MSCAN_TXEMPTY_ALL_INTS	None	Enables or disables the transmitter-empty interrupts for the selected transmit buffers.
MSCAN_GET_ENABLED_TINT	NULL	UWord16	Returns bit-mask of transmit buffer where interrupt is enabled.
MSCAN_READ_ERINT_FLAGS	NULL	UWord16	Reads and returns the CAN Error or Receiver interrupt flags.
MSCAN_READ_EINT_FLAGS	NULL	UWord16	Reads and returns the CAN Error interrupt flags.
MSCAN_READ_TINT_FLAGS	NULL	UWord16	Reads and returns the CAN Transmitter flags.
MSCAN_CLEAR_ERINT_FLAGS	UWord16: Error and Receiver interrupt flags to clear: MSCAN_WAKEUP_INT | MSCAN_STATCHNG_INT | MSCAN_OVERRUN_INT | MSCAN_RXFULL_INT	None	Clears the selected CAN Error or Receiver interrupts.
MSCAN_CLEAR_EINT_FLAGS	UWord16: Error interrupt flags to clear: MSCAN_STATCHNG_INT | MSCAN_OVERRUN_INT	None	Clears the selected CAN Error interrupts. Typically used in the CAN Error interrupt service routine with parameter obtained from: MSCAN_READ_EINT_FLAGS

Table 5-858. ioctl commands (Continued)

Cmd	param	Return	Description
MSCAN_CLEAR_RINT_FLAG	NULL	None	Clears the CAN Receiver-full interrupt flag. Typically used in the CAN Receiver interrupt service routine.
MSCAN_CLEAR_WINT_FLAG	NULL	None	Clears the CAN Wake-up interrupt flag. Typically used in the CAN Wake-up interrupt service routine.
MSCAN_SELECT_TXBUFF	Any combination of: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2 MSCAN_ANY_TXBUFFER may also be used (combination of all flags above).	UWord16 zero/non-zero	Maps (makes accessible) one of buffers in the CAN peripheral space. Returns the flag which identifies the selected buffer which is empty and ready to be filled. No buffer is available when zero is returned.
MSCAN_SELECT_NEXT_TXBUFF	NULL	UWord16 buffer bit or zero	Tries to find a free transmit buffer, maps it into the CAN peripheral space and returns its identification bit. Returns zero if no buffer is available.
MSCAN_TRANSMIT	Any combination of: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2	None	Schedules given buffer(s) for transmission. Typically a value returned from MSCAN_SELECT _NEXT_TXBUFF is used as a parameter.
MSCAN_ABORT_TRANSMIT	Any combination of: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2	None	De-schedules selected buffers from transmission, if transmission is still pending.
MSCAN_TEST_BUSOFF_HOLD	NULL	UWord16 zero/non-zero	Tests if bus-off condition needs to be serviced.
MSCAN_RECOVER_BUSOFF_STATE	NULL	None	Services the bus-off condition.
MSCAN_GET_WINNING_ ACC_FILTER	NULL	UWord16 (0..7)	Tests what acceptance filter was used when currently received frame was being evaluated.
MSCAN_GET_RX_ERR_COUNT	NULL	UWord16	Reads the RX error counter register.
MSCAN_GET_TX_ERR_COUNT	NULL	UWord16	Reads the TX error counter register.

Table 5-859. Message Buffer ioctl commands

Cmd	param	Return	Description
MSCANMB_GET_ID	NULL	UWord32	Parses 29+1bit ID from the ID-raw registers of MB. Returns as 32bit number where MSB signals an Extended ID.
MSCANMB_GET_ID_RAW	NULL	UWord32	Returns four ID-raw registers of MB as a single 32 bit word.
MSCANMB_GET_LEN	NULL	UWord32	Returns the Data Length of the specified MB.
MSCANMB_GET_RTR	NULL	UWord16	Returns non-zero if RTR is set in given MB.
MSCANMB_GET_DATAPTR	NULL	UWord16 *	Returns the pointer to the Data part of the specified MB structure.
MSCANMB_GET_TIMESTAM	NULL	UWord16	Returns the time-stamp value attached to the message buffer.
MSCANMB_SET_ID	32bit Id (constant) Combine with MSCAN_ID_EXT when setting extended ID. Combine with MSCAN_ID_RTR when sending a Remote Transmit Request.	None	Calculates the ID-raw value and writes it to the appropriate registers of given MB (use when parameter is a constant).
MSCANMB_SET_ID_V	32bit Id (variable) Combine with MSCAN_ID_EXT when setting extended ID. Combine with MSCAN_ID_RTR when sending a Remote Transmit Request.	None	Same as above but implemented as a function call (more suitable when passing a variable value).
MSCANMB_SET_ID_RAW	32bit value	None	Sets all four IDR registers from a single 32 bit word.
MSCANMB_SET_RTR	MSCAN_ON/ MSCAN_OFF	None	Sets or clears the RTR bit in given MB. Should be used after the ID is written to MB.
MSCANMB_SET_LEN	[0,8]	None	Sets the length field of the MB.
MSCANMB_SET_TBP	[0,255]	None	Sets Transmit Buffer Priority of the MB (the lower value, the higher priority).

5.25.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.25.3.1 MSCAN_INIT - initialize the MSCAN module

Call(s):

```
void ioctl(const int *pModuleBase, MSCAN_INIT, NULL);
```

Arguments:

Table 5-860. MSCAN_INIT ioctl call arguments

*pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_INIT* ioctl command executes the initialization function, which initializes all configured MSCAN registers by the values defined in *appconfig.h*. It is intended for the static configuration of the MSCAN module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_INIT* ioctl command is implemented as a function call. Only necessary MSCAN registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-748. MSCAN_INIT

```
ioctl(MSCAN, MSCAN_INIT, NULL);
```

This code initializes the MSCAN module by the values defined in the *appconfig.h* file. The *appconfig.h* file can be edited manually or generated by the Graphical Configuration Tool.

5.25.3.2 MSCAN_SOFT_RESET - enter or leave the soft reset state

Call(s):

```
void ioctl(const int *pModuleBase, MSCAN_SOFT_RESET,
     UWord16 param);
```

Arguments:

Table 5-861. MSCAN_SOFT_RESET ioctl call arguments

*pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enter the soft reset state MSCAN_DISABLE - leave the soft reset state

Description: The *MSCAN_SOFT_RESET* ioctl command enters or leaves the internal soft-reset state of the MSCAN module depending on the value of the argument. In case of its enabling, any ongoing transmission or reception is quit and synchronization to the bus is lost.

The following registers enter and stay in their hard Reset state: CANCTL0, CANRFLG, CANRIER, CANTFLG, and CANTCR. The registers CANCTL1, CANBTR0, CANBTR1, CANIDAC, CANIDAR0–7, and CANIDMR0–7 can only be modified by the CPU when the MSCAN is in the soft reset state. Values of the error counters are not affected by the soft reset.

This command writes directly into the INITRQ (formerly SFTRES) bit of the CANCTL0 register and waits until the soft-reset state is acknowledged by the INITAK bit of the CANCTL1 register.

See the MSCAN module documentation for complete description of the soft-reset effects.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_SOFT_RESET* ioctl command is implemented as a macro.

Example 5-749. MSCAN_SOFT_RESET

```
ioctl(MSCAN, MSCAN_SOFT_RESET, MSCAN_ENABLE);
```

This code activates the soft reset state of the MSCAN module.

5.25.3.3 MSCAN_DEVICE - enable or disable the MSCAN device

Call(s):

```
void ioctl(const int *pModuleBase, MSCAN_DEVICE, UWord16 param);
```

Arguments:

Table 5-862. MSCAN_DEVICE ioctl call arguments

*pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable the MSCAN device MSCAN_DISABLE - disable the MSCAN device

Description: This command enables or disables the MSCAN module.

This command directly sets or clears the CANE bit of the CANCTL1 register.

Returns: None.

Range Issues: None

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_DEVICE* command is implemented as a macro.

Example 5-750. MSCAN_DEVICE

```
ioctl(MSCAN, MSCAN_DEVICE, MSCAN_ENABLE);
```

This code enables the MSCAN module.

5.25.3.4 MSCAN_WAKEUP_FILTER - enable or disable the low-pass filter on wake-up condition

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_WAKEUP_FILTER,
     UWord16 param);
```

Arguments:

Table 5-863. MSCAN_WAKEUP_FILTER ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable low-pass filter on wake-up pin MSCAN_DISABLE - disable low-pass filter on wake-up pin

Description: This command defines whether the integrated low-pass filter is applied to protect the MSCAN from spurious wake-up. When enabled, the CAN module wakes-up the CPU only in case of a dominant pulse on the CAN bus having a length of T_{WUP}.

This command directly sets or clears the WUPM bit of the CANCTL1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_WAKEUP_FILTER* ioctl command is implemented as a macro.

Example 5-751. MSCAN_WAKEUP_FILTER

```
ioctl(MSCAN, MSCAN_WAKEUP_FILTER, MSCAN_ENABLE);
```

This code enables the MSCAN low-pass filter on wake-up pin.

5.25.3.5 MSCAN_MANUAL_BOFF_RECOVERY - enable or disable manual bus-off condition recovery

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_MANUAL_BOFF_RECOVERY,
     UWord16 param);
```

Arguments:

Table 5-864. MSCAN_MANUAL_BOFF_RECOVERY ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable manual bus-off recovery MSCAN_DISABLE - disable manual bus-off recovery

Description: This command selects how the MSCAN module handles recovery from a bus-off condition. When manual recovery is disabled, an automatic bus-off recovery is performed according to the CAN 2.0A/B specification. When manual recovery is enabled, the application code has to handle or poll the bus-off error interrupt and use the MSCAN_RECOVER_BUSOFF_STATE ioctl command to recover from the bus-off condition.

This command directly sets or clears the BORM bit of the CANCTL1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_MANUAL_BOFF_RECOVERY* ioctl command is implemented as a macro.

Example 5-752. MSCAN_MANUAL_BOFF_RECOVERY

```
ioctl(MSCAN, MSCAN_MANUAL_BOFF_RECOVERY, MSCAN_ENABLE);
```

This code enables the manual bus-off recovery mode.

5.25.3.6 MSCAN_LISTEN_ONLY_MODE - enable or disable listen only mode

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_LISTEN_ONLY_MODE,
     UWord16 param);
```

Arguments:

Table 5-865. MSCAN_LISTEN_ONLY_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable listen-only mode MSCAN_DISABLE - disable listen-only mode

Description: This command enables or disables the listen-only mode of the MSCAN module operation. In the listen-only mode, all valid CAN messages with matching ID are received, but no acknowledgement or error frames are sent out. Additionally, the error counters are frozen. The Listen-only mode supports applications requiring hot-plugging, or throughput analysis. The CAN is unable to transmit any messages when listen-only mode is active.

This command directly sets or clears the LISTEN bit of the CANCTL1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_LISTEN_ONLY_MODE* ioctl command is implemented as a macro.

Example 5-753. MSCAN_LISTEN_ONLY_MODE

```
ioctl(MSCAN, MSCAN_LISTEN_ONLY_MODE, MSCAN_ENABLE);
```

This code enables the listen-only mode.

5.25.3.7 MSCAN_LOOPBACK_MODE - enable or disable test loopback mode

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_LOOPBACK_MODE,
     UWord16 param);
```

Arguments:

Table 5-866. MSCAN_LOOPBACK_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable test loopback mode MSCAN_DISABLE - disable test loopback mode

Description: This command enables or disables the test loopback mode. When enabled, the MSCAN performs an internal loop back used for a self test operation. The bit stream output of the transmitter is fed back to the receiver internally. The MSCAN_RX input pin is ignored while the MSCAN_TX output goes to the recessive state, logic one. The MSCAN behaves as it does when normally transmitting. It treats its own transmitted message as a message received from a remote node. In this state, the MSCAN ignores the bit sent during the ACK slot in the CAN frame Acknowledge field ensuring proper reception of its own message. Both transmit and receive interrupts are generated.

This command directly sets or clears the LOOPB bit of the CANCTL1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_LOOPBACK_MODE* ioctl command is implemented as a macro.

Example 5-754. MSCAN_LOOPBACK_MODE

```
ioctl(MSCAN, MSCAN_LOOPBACK_MODE, MSCAN_ENABLE);
```

This code enables the MSCAN test loopback mode.

5.25.3.8 MSCAN_SET_CLOCK_SOURCE - select clock source

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_CLOCK_SOURCE,
     UWord16 param);
```

Arguments:

Table 5-867. MSCAN_SET_CLOCK_SOURCE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_IPBUS - choose the IPBUS clock as CANCLK MSCAN_XTAL - choose the crystal oscillator as CANCLK

Description: This command selects the clock source to be used as the CANCLK clock. Either the IPBus clock or crystal oscillator clock source may be selected.

This command directly sets or clears the CLKSRC bit of the CANCTL1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- If the IPbus clock is generated from a PLL, then selecting the crystal clock source from 128Kbits/sec up to 500Kbits/sec will result in the least amount of clock jitter and the most reliable data transfer. When using the IPbus clock, up to 1Mbits/sec is possible although transfer will be slightly less reliable due to clock jitter; however, it is still well within the requirements of the CAN bus specification.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_CLOCK_SOURCE* ioctl command is implemented as a macro.

Example 5-755. MSCAN_SET_CLOCK_SOURCE

```
ioctl(MSCAN, MSCAN_SET_CLOCK_SOURCE, MSCAN_XTAL);
```

This code sets the crystal oscillator to serve as the CANCLK clock.

5.25.3.9 MSCAN_SET_PRESCALER - set prescaler divide factor

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-868. MSCAN_SET_PRESCALER ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	The value to set the prescaler to (range of 1..64)

Description: The *MSCAN_SET_PRESCALER* ioctl command sets the ratio between the CANCLK signal frequency and the Time quanta clock frequency of the MSCAN module.

Returns: None.

Range Issues: Only values 1 to 64 are allowed.

Special Issues:

- This command is applicable only on MC56F803x.
- The resulting baud rate of the MSCAN module depends on several other parameters in the CANBTR0 and CANBTR1 registers. Section 5.25.2.1 contains more information about this topic.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_PRESCALER* ioctl command is implemented as a macro.

Example 5-756. MSCAN_SET_PRESCALER

```
ioctl(MSCAN, MSCAN_SET_CLOCK_SOURCE, MSCAN_XTAL);
ioctl(MSCAN, MSCAN_SET_PRESCALER, 8);
```

This code sets the clock divisor to 8. In the case where the clock source has the frequency 8 MHz, one time quantum will be equal to 1 us.

5.25.3.10 MSCAN_SET_SJW - set Synchronization Jump Width

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_SJW, UWord16 param);
```

Arguments:

Table 5-869. MSCAN_SET_RJW ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Number of time quanta to set the as the Synchronization Jump Width parameter. Use the following predefined constants or their UWord16 equivalents in the range 1..4: MSCAN_SJW_1... MSCAN_SJW_4

Description: The *MSCAN_SET_SJW* ioctl command sets the Synchronization Jump Width parameter of the CAN bit timing machine. The SJW is specified in the time-quanta units and is always at least 1 tq. See the CAN standard document for more information.

Returns: None.

Range Issues: Only numerical values from 1 to 4 are allowed. Use the *MSCAN_SJW_n* constants defined in the *mscan.h* file.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_SJW* ioctl command is implemented as a macro.

Example 5-757. MSCAN_SET_SJW

```
ioctl(MSCAN, MSCAN_SET_SJW, MSCAN_SJW_1);
```

This code sets the Synchronization Jump Width value to 1.

5.25.3.11 MSCAN_SET_TSEG1 - set Time Segment 1

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_TSEG1, UWord16 param);
```

Arguments:

Table 5-870. MSCAN_SET_TSEG1 ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Number of time quanta to set as the Time Segment 1 parameter. Use the following predefined constants or their UWord16 equivalents in the range 4..16: MSCAN_TSEG_4... MSCAN_TSEG_16

Description: The *MSCAN_SET_TSEG1* ioctl command sets the Time Segment 1 parameter of the CAN bit timing machine. The Time Segment 1 is specified in time-quanta units and is always at least 1 tq. See the CAN standard document for more information.

Returns: None.

Range Issues: Only numerical values from 4 to 16 are allowed. Use the *MSCAN_TSEG_n* constants defined in the *mscan.h* file.

Special Issues:

- This command is applicable only on MC56F803x.
- The resulting baud rate of the MSCAN module depends on several other parameters in the CANBTR0 and CANBTR1 registers. Section 5.25.2.1 contains more information about this topic.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_TSEG1* ioctl command is implemented as a macro.

Example 5-758. MSCAN_SET_TSEG1

```
ioctl(MSCAN, MSCAN_SET_TSEG1, MSCAN_TSEG_5);
```

This code sets the Time Segment 1 parameter to 5.

5.25.3.12 MSCAN_SET_TSEG2 - set Time Segment 2

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_TSEG2, UWord16 param);
```

Arguments:

Table 5-871. MSCAN_SET_TSEG2 ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Number of time quanta to set as the Time Segment 2 parameter. Use the following predefined constants or their UWord16 equivalents in the range 2..8: MSCAN_TSEG_2... MSCAN_TSEG_8

Description: The *MSCAN_SET_TSEG2* ioctl command sets the Time Segment 2 parameter of the CAN bit timing machine. The Time Segment 2 is specified in time-quanta units and is always at least 1 tq. See the CAN standard document for more information.

Returns: None.

Range Issues: Only numerical values from 2 to 8 are allowed. Use the *MSCAN_TSEG_n* constants defined in the *mscan.h* file.

Special Issues:

- This command is applicable only on MC56F803x.
- The resulting baud rate of the MSCAN module depends on several other parameters in the CANBTR0 and CANBTR1 registers. Section 5.25.2.1 contains more information about this topic.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_TSEG2* ioctl command is implemented as a macro.

Example 5-759. MSCAN_SET_TSEG2

```
ioctl(MSCAN, MSCAN_SET_TSEG2, MSCAN_TSEG_3);
```

This code sets the Time Segment 2 parameter to 3.

5.25.3.13 MSCAN_SET_SAMPLING - set bit-sampling mode

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_SAMPLING,
           UWord16 param);
```

Arguments:

Table 5-872. MSCAN_SET_SAMPLING ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Parameter to select the desired sampling mode. Use one of these predefined constants: MSCAN_1SAMP_PER_BIT - one sample per bit MSCAN_3SAMPS_PER_BIT - three samples per bit

Description: The *MSCAN_SET_SAMPLING* ioctl command configures the receiving machine and selects how many times each bit is sampled. The possible values are “once per bit” (reset default) or “three times per bit”. Three times sampling includes the regular sample point and two preceding samples and uses a majority rule to determine the resulting bit value.

This command directly sets or clears the SAMP bit of the CANBTR1 register.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_SAMPLING* ioctl command is implemented as a macro.

Example 5-760. MSCAN_SET_SAMPLING

```
ioctl(MSCAN, MSCAN_SET_SAMPLING, MSCAN_3SAMPS_PER_BIT);
```

This code selects the three-samples-per-bit sampling mode.

5.25.3.14 MSCAN_SET_ACC_MODE - set acceptance filter mode

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MODE,
     UWord16 param);
```

Arguments:

Table 5-873. MSCAN_SET_ACC_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Parameter to select the desired acceptance filter configuration. Use one of these predefined constants: MSCAN_ACC_MODE_2X32 - two 32-bit Acceptance Filters MSCAN_ACC_MODE_4X16 - four 16-bit Acceptance Filters MSCAN_ACC_MODE_8X8 - eight 8-bit Acceptance Filters MSCAN_ACC_MODE_CLOSED - Filter Closed

Description: The *MSCAN_SET_ACC_MODE* ioctl command configures the receive acceptance filter, which is applied any time a new message is received.

This command directly accesses the IDAM0 and IDAM1 bits of the CANIDAC register.

See the MSCAN module documentation for a more detailed description of the acceptance filtering.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_MODE* ioctl command is implemented as a macro.

Example 5-761. MSCAN_SET_ACC_MODE

```
ioctl(MSCAN, MSCAN_SET_ACC_MODE, MSCAN_ACC_MODE_2X32);
```

This code selects the acceptance mode with two 32-bit Acceptance Filters.

5.25.3.15 MSCAN_SET_ACC_MASKR_32_*x* - set 32-bit acceptance filter mask for filter *x (0..1)*

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_32_0,
     UWord32 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_32_1,
     UWord32 param);
```

Arguments:

Table 5-874. MSCAN_SET_ACC_MASKR_32_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	UWord32 as a 32-bit ID-raw value to be used as a mask for the acceptance filtering

Description: The *MSCAN_SET_ACC_MASKR_32_x* ioctl commands configure the acceptance filter mask for two filters 0 and 1. The acceptance filter must first be switched to the 32-bit mode, using the MSCAN_SET_ACC_MODE ioctl command with the MSCAN_ACC_MODE_2X32 parameter.

Each bit in the filter mask value determines if the same bit in a received message's ID-raw value *must* match bit value in the acceptance filter value or if this bit is *not* compared.

- Logic *one*s in the mask value identify "don't care" bits, which means the matching ID-raw bits of the received message are *not* compared with bits in the acceptance value.
- Logic *zero*s in the mask value identify bits which must match the acceptance value, in order to successfully receive the CAN message.

Be aware that the ID-raw value of the received message is evaluated by the filter, which means the filter may also be designed to evaluate the IDE, RTR/SRR and RTR bits of the received CAN message. Also be aware that bits [18:16] of the 32-bit value must be set to 1 (don't care) in the mask register when the standard 11-bit CAN messages are to be filtered. Bits [15:0] of the 32-bit mask value are ignored when a standard message is received.

This command directly accesses the CANIDMR0-3 registers for filter 0 and the CANIDMR4-7 registers for filter 1.

See the MSCAN module documentation for a more detailed description of acceptance filtering.

Returns: None.

Range Issues: See description above.

Special Issues: This command is applicable only on MC56F803x. This command has an effect only when MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_MASKR_32_x* ioctl commands are both implemented as macros.

Example 5-762. MSCAN_SET_ACC_MASKR_32_*x*

```
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_32_0, 0x0003FFFF);
```

This code sets the acceptance mask register 0 to evaluate all ID-raw bits of the standard CAN identifiers. Low-order bits are set to 1 as "don't care".

5.25.3.16 MSCAN_SET_ACC_MASKR_16_*x* - set 16-bit acceptance filter mask for filter *x (0..3)*

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_16_0,
     UWord16 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_16_1,
     UWord16 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_16_2,
     UWord16 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_16_3,
     UWord16 param);
```

Arguments:

Table 5-875. MSCAN_SET_ACC_MASKR_16_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	UWord16 as a 16-bit ID-raw value to be used as a mask for the acceptance filtering

Description: The *MSCAN_SET_ACC_MASKR_16_x* ioctl commands configure the acceptance filter masks for four filters 0, 1, 2 and 3. The acceptance filter must first be switched to the 16-bit mode using the MSCAN_SET_ACC_MODE ioctl command with the MSCAN_ACC_MODE_4X16 parameter.

Please see the mask value description above in the MSCAN_SET_ACC_MASKR_32_x ioctl command documentation.

Be aware that the ID-raw value of the received message is evaluated by the filter, which means the filter may also be designed to evaluate the IDE, RTR/SRR and RTR bits of the received CAN message. Also be aware that bits [2:0] of the 16-bit value must be set to 1 (don't care) in the mask register when the standard 11-bit CAN messages are to be filtered.

When the extended CAN IDs are to be evaluated in the 16-bit filter mode, be aware that the 16-bit mask value matches the CAN ID bits [28:15] only (with the IDE and SRR bits on bit-positions 4 and 3).

This command directly accesses the CANIDMR0-1 registers for filter 0, CANIDMR2-3 registers for filter 1, CANIDMR4-5 registers for filter 2 and CANIDMR6-7 registers for filter 3.

Returns: None.

Range Issues: See description above.

Special Issues: This command is applicable only on MC56F803x. This command has an effect only when MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_MASKR_16_x* ioctl commands are all implemented as macros.

Example 5-763. MSCAN_SET_ACC_MASKR_16_*x*

```
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_16_0, 0x0003);
```

This code sets the acceptance mask register 0 to evaluate all ID-raw bits of the standard CAN identifiers. Low-order bits need to be set to 1 as "don't care".

5.25.3.17 MSCAN_SET_ACC_MASKR_8_*x* - set 8-bit acceptance filter mask for filter *x (0..7)*

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_8_0,
     UWord16 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_8_1,
     UWord16 param);

... dtto for filters 2, 3, 4, 5, 6 ...

void ioctl(const int pModuleBase, MSCAN_SET_ACC_MASKR_8_7,
     UWord16 param);
```

Arguments:

Table 5-876. MSCAN_SET_ACC_MASKR_8_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	UWord16 as a ID-raw value to be used as a mask for the acceptance filtering. Only eight low-order bits of the parameter value are used.

Description: The *MSCAN_SET_ACC_MASKR_8_x* ioctl commands configure the acceptance filter mask for eight filters, 0 to 7. The acceptance filter must first be switched to the 8-bit mode, using the MSCAN_SET_ACC_MODE ioctl command with the MSCAN_ACC_MODE_8X8 parameter.

Please see the mask value description above in the MSCAN_SET_ACC_MASKR_32_x ioctl command documentation.

Be aware that the 8-bit filter mask value matches the bits [10:3] of the standard CAN ID and bits [28:21] of the extended CAN ID values. It is not possible to evaluate the special IDE, SRR/RTR and RTR bits in the 8-bit filter mode.

This command directly accesses the CANIDMR*x* register for each filter *x*.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_MASKR_8_x* ioctl commands are all implemented as macros.

Example 5-764. MSCAN_SET_ACC_MASKR_8_*x*

```
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_8_0, 0x00);
```

This code sets the acceptance mask register 0 to evaluate all ID-raw bits of the standard CAN identifiers.

5.25.3.18 MSCAN_SET_ACC_IDR_32_*x* - set 32-bit acceptance filter value for filter *x* (0..1)

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_32_0,
     UWord32 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_32_1,
     UWord32 param);
```

Arguments:

Table 5-877. MSCAN_SET_ACC_IDR_32_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	32-bit value to be used for the acceptance filtering.

Description: The *MSCAN_SET_ACC_IDR_32_x* ioctl commands configure the acceptance filtering values for the 32-bit filters 0 and 1. The selected bits of the filter values are compared with matching bits of the ID-raw value of each received CAN message to determine if a message is to be saved in a receive buffer.

See more details about the 32-bit acceptance filter mode above in the description of the MSCAN_SET_ACC_MASKR_32_*x* ioctl commands.

This command directly accesses the CANIDAR0-3 registers for filter 0 and CANIDAR4-7 registers for filter 1.

See the MSCAN module documentation for a more detailed description.

Returns: None.

Range Issues: See the MSCAN_SET_ACC_MASKR_32_*x* ioctl command description.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_IDR_32_x* ioctl commands are both implemented as macros.

Example 5-765. MSCAN_SET_ACC_IDR_32_x

```
ioctl(MSCAN, MSCAN_SET_ACC_MODE, MSCAN_ACC_MODE_2X32);
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_32_0, 0x00000001);
ioctl(MSCAN, MSCAN_SET_ACC_IDR_32_0, 0x91B8ACF8);
```

This code sets the acceptance filter 0 to accept exclusively the extended CAN messages (IDE=1) with the exact CAN ID of 0x12345678. The SRR bit is required to be 1, the RTR bit is not evaluated so both RTR and non-RTR messages are received.

5.25.3.19 MSCAN_SET_ACC_IDR_16_*x* - set 16-bit acceptance filter value for filter *x* (0..3)

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_16_0,
     UWord16 param);

... dtto for filters 1, 2, 3 ...

void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_16_3,
     UWord16 param);
```

Arguments:

Table 5-878. MSCAN_SET_ACC_IDR_16_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	UWord16 as 16-bit value to be used for the acceptance filtering.

Description: The *MSCAN_SET_ACC_IDR_16_x* ioctl commands configure the acceptance filtering values for the 16-bit filters 0, 1, 2 and 3. The selected bits of the filter values are compared with matching bits of the ID-raw value of each received CAN message to determine if a message is to be saved in a receive buffer.

See more details about the 16-bit acceptance filter mode above in the description of the MSCAN_SET_ACC_MASKR_16_*x* ioctl commands.

This command directly accesses the CANIDAR0-1 registers for filter 0, CANIDAR2-3 registers for filter 1, CANIDAR4-5 registers for filter 2 and CANIDAR6-7 registers for filter 3.

See the MSCAN module documentation for a more detailed description.

Returns: None.

Range Issues: See the MSCAN_SET_ACC_MASKR_16_*x* ioctl command description.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_IDR_16_x* ioctl commands are all implemented as macros.

Example 5-766. MSCAN_SET_ACC_IDR_16_x

```
ioctl(MSCAN, MSCAN_SET_ACC_MODE, MSCAN_ACC_MODE_4X16);
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_16_0, 0x0013);
ioctl(MSCAN, MSCAN_SET_ACC_IDR_16_0, 0x2460);
```

This code sets the acceptance filter 0 to accept exclusively the standard CAN messages (IDE=0) with the exact CAN ID of 0x123. The RTR bit is not evaluated so both RTR and non-RTR messages are received. Note that bits [2:0] are set as “don’t care” in the mask, which is required for the standard ID filtering in the 16-bit filter mode.

5.25.3.20 MSCAN_SET_ACC_IDR_8_*x* - set 8-bit acceptance filter value for filter *x* (0..7)

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_8_0,
     UWord16 param);
void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_8_1,
     UWord16 param);

... dtto for filters 2, 3, 4, 5, 6 ...

void ioctl(const int pModuleBase, MSCAN_SET_ACC_IDR_8_7,
     UWord16 param);
```

Arguments:

Table 5-879. MSCAN_SET_ACC_IDR_8_*x* ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	UWord16 as 16-bit value to be used for acceptance filtering. Only eight low-order bits of the parameter value are used.

Description: The *MSCAN_SET_ACC_IDR_8_x* ioctl commands configure the acceptance filtering values for the 8-bit filters 0 to 7. The selected bits of the filter values are compared with matching bits of the ID-raw value of each received CAN message to determine if a message is to be saved in a receive buffer.

See more details about the 8-bit acceptance filter mode above in the description of the MSCAN_SET_ACC_MASKR_8_*x* ioctl commands.

This command directly accesses the CANIDAR*x* registers for each filter *x*.

See the MSCAN module documentation for a more detailed description.

Returns: None.

Range Issues: See the MSCAN_SET_ACC_MASKR_8_*x* ioctl command description.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in soft-reset (initialization) state.

Design/Implementation: The *MSCAN_SET_ACC_IDR_8_x* ioctl commands are all implemented as macros.

Example 5-767. MSCAN_SET_ACC_IDR_8_x

```
ioctl(MSCAN, MSCAN_SET_ACC_MODE, MSCAN_ACC_MODE_8X8);
ioctl(MSCAN, MSCAN_SET_ACC_MASKR_8_0, 0x00);
ioctl(MSCAN, MSCAN_SET_ACC_IDR_8_0, 0x24);
```

This code sets the acceptance filter 0 to accept the standard messages where the CAN ID match the value 0x120. Three low-order bits of the CAN ID are not compared so the CAN messages of IDs 0x120 to 0x127 are also accepted by this filter. As the 8-bit filter can not cover the IDE bit of the ID-raw value, the extended identifiers matching the values 0x04800000 to 0x49FFFFFF would also be accepted.

5.25.3.21 MSCAN_SLEEP - enter or leave the sleep state

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_SLEEP, UWord16 param);
```

Arguments:

Table 5-880. MSCAN_SLEEP ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ON - enter the sleep state MSCAN_OFF - leave the sleep state

Description: The *MSCAN_SLEEP* ioctl command enters or leaves the sleeping state of the MSCAN module depending on the value of the argument.

This command sets or resets directly the SLPRQ bit of the CANCTL0 register. If sleep mode is requested, this command waits for the sleep-mode acknowledge SLPAK bit of the CANCTL1 register.

See the MSCAN module documentation for complete description of the sleep mode.

Returns: None.

Range Issues: None.

Special Issues:

- This command is applicable only on MC56F803x.
- This command has an effect only when the MSCAN module is in operational mode and soft-reset is de-asserted.

Design/Implementation: The *MSCAN_SLEEP* ioctl command is implemented as a macro.

Example 5-768. MSCAN_SLEEP

```
ioctl(MSCAN, MSCAN_SLEEP, MSCAN_ENABLE);
```

This code forces the MSCAN module to enter the sleep mode.

5.25.3.22 MSCAN_GET_SLEEP_MODE - returns the sleeping state

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_GET_SLEEP_MODE, NULL);
```

Arguments:

Table 5-881. MSCAN_GET_SLEEP_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_GET_SLEEP_MODE* ioctl command returns a non-zero value if the MSCAN is in the sleeping mode. This command returns zero when the MSCAN module is active (not-sleeping).

This command tests the status of the SLPRQ and SLPAK bits of the CANCTL0 register.

See the MSCAN module documentation for complete description of the sleep mode.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_GET_SLEEP_MODE* ioctl command is implemented as a macro.

Example 5-769. MSCAN_GET_SLEEP_MODE

```
if(ioctl(MSCAN, MSCAN_GET_SLEEP_MODE, NULL))
{
      // MSCAN module is sleeping
}
```

This code test the MSCAN sleep state.

5.25.3.23 MSCAN_AUTO_WAKEUP - enable or disable automatic wake-up

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_AUTO_WAKEUP,
     UWord16 param);
```

Arguments:

Table 5-882. MSCAN_AUTO_WAKEUP ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable automatic wake-up MSCAN_DISABLE - disable automatic wake-up

Description: The *MSCAN_AUTO_WAKEUP* ioctl command enables or disables the automatic wake-up feature of the MSCAN module. When enabled, a traffic detected on the CAN bus may wake the MSCAN module out of the sleeping state. See also the MSCAN_WAKEUP_FILTER ioctl command for more details about this feature.

This command sets or clears directly the WUPE bit of the CANCTL0 register.

See the MSCAN module documentation for complete description of the sleep mode.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_AUTO_WAKEUP* ioctl command is implemented as a macro.

Example 5-770. MSCAN_AUTO_WAKEUP

```
ioctl(MSCAN, MSCAN_AUTO_WAKEUP, MSCAN_ENABLE);
ioctl(MSCAN, MSCAN_SLEEP, MSCAN_ON);
```

This code enables the automatic wake-up and then it puts the MSCAN module into the sleeping state.

5.25.3.24 MSCAN_TIMESTAMP_TIMER - enable or disable time-stamp timer

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_TIMESTAMP_TIMER,
     UWord16 param);
```

Arguments:

Table 5-883. MSCAN_TIMESTAMP_TIMER ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - enable time-stamp timer MSCAN_DISABLE - disable time-stamp timer

Description: The *MSCAN_TIMESTAMP_TIMER* ioctl command enables or disables the internal time-stamp timer. When enabled, this timer-counter runs at speed of the MSCAN bit-clock rate and its value is captured each time a CAN message is received or transmitted (acknowledged). The captured value is then saved to the receive or transmit message buffer. When disabled the timer-counter is reset and all time-stamp values are captured and stored as zero.

This command sets or clears directly the TIME bit of the CANCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_TIMESTAMP_TIMER* ioctl command is implemented as a macro.

Example 5-771. MSCAN_TIMESTAMP_TIMER

```
ioctl(MSCAN, MSCAN_TIMESTAMP_TIMER, MSCAN_ENABLE);
```

This code enables the timer and time-stamps values to be generated.

5.25.3.25 MSCAN_TEST_SYNCH - test if synchronized to CAN bus

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_TEST_SYNCH, NULL);
```

Arguments:

Table 5-884. MSCAN_TEST_SYNCH ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_TEST_SYNCH* ioctl command returns a non-zero value if the MSCAN module is synchronized to the CAN bus. When synchronized, it is able to participate in the communication process.

This command tests the status of the SYNCH of the CANCTL0 register.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCAN_TEST_SYNCH* ioctl command is implemented as a macro.

Example 5-772. MSCAN_TEST_SYNCH

```
if(ioctl(MSCAN, MSCAN_TEST_SYNCH, NULL))
{
      // synchronized to CAN bus
}
```

This code checks if MSCAN module is synchronized to the bus.

5.25.3.26 MSCAN_TEST_RXACT - test if receiving is in progress

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_TEST_RXACT, NULL);
```

Arguments:

Table 5-885. MSCAN_TEST_RXACT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_TEST_RXACT* ioctl command returns a non-zero value if the MSCAN module is just receiving the CAN message.

This command tests the status of the RXACT of the CANCTL0 register.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None.

Special Issues: This command does not return valid result when MSCAN is in loop-back mode.

Design/Implementation: The *MSCAN_TEST_RXACT* ioctl command is implemented as a macro.

Example 5-773. MSCAN_TEST_RXACT

```
while(ioctl(MSCAN, MSCAN_TEST_RXACT, NULL))
{
      ; // wait until current message is received
}
```

This code spins until the message which is currently being received is received completely.

5.25.3.27 MSCAN_TEST_RXFRM - test if frame was received

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_TEST_RXFRM, NULL);
```

Arguments:

Table 5-886. MSCAN_TEST_RXFRM ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_TEST_RXFRM* ioctl command returns a non-zero value if a CAN frame was received since the MSCAN_CLEAR_RXFRM ioctl command was called last time. The RXFRM flag can not be used to generate the CPU interrupt. Do not confuse the RXFRM flag with the RXFULL interrupt flag.

This command tests the status of the RXFRM of the CANCTL0 register.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCAN_TEST_RXFRM* ioctl command is implemented as a macro.

Example 5-774. MSCAN_TEST_RXFRM

```
while(ioctl(MSCAN, MSCAN_TEST_RXFRM, NULL))
{
      ; // a new message received

      ioctl(MSCAN, MSCAN_CLEAR_RXFRM, NULL);
}
```

This code checks if a new message was received.

5.25.3.28 MSCAN_CLEAR_RXFRM - acknowledge frame received flag

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_CLEAR_RXFRM, NULL);
```

Arguments:

Table 5-887. MSCAN_CLEAR_RXFRM ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_CLEAR_RXFRM* ioctl command clears the RXFRM flag so the next frame will be recognized using the MSCAN_TEST_RXFRM ioctl command. The RXFRM flag is not used to generate the CPU interrupt. Do not confuse clearing the RXFRM flag with clearing if the RXFULL interrupt flag.

This command writes one to the RXFRM of the CANCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCAN_CLEAR_RXFRM* ioctl command is implemented as a macro.

Example 5-775. MSCAN_CLEAR_RXFRM

```
while(ioctl(MSCAN, MSCAN_TEST_RXFRM, NULL))
{
      ; // a new message received

      ioctl(MSCAN, MSCAN_CLEAR_RXFRM, NULL);
}
```

This code checks if a new message was received.

5.25.3.29 MSCAN_STOP_IN_WAIT - enable or disable MSCAN shutdown during the CPU wait mode

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_STOP_IN_WAIT,
     UWord16 param);
```

Arguments:

Table 5-888. MSCAN_STOP_IN_WAIT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	MSCAN_ENABLE - MSCAN will shutdown during the CPU wait mode. MSCAN_DISABLE - MSCAN module will not be affected by the CPU wait mode.

Description: The *MSCAN_STOP_IN_WAIT* ioctl command enables or disables suspending to a power-saving shutdown mode when the CPU goes to the wait mode.

This command sets or clears directly the CSWAI bit of the CANCTL0 register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_STOP_IN_WAIT* ioctl command is implemented as a macro.

Example 5-776. MSCAN_STOP_IN_WAIT

```
ioctl(MSCAN, MSCAN_STOP_IN_WAIT, MSCAN_DISABLE);
```

This code disables the low-power suspend mode. MSCAN will not be affected by the CPU wait mode.

5.25.3.30 MSCAN_ERINT_ENABLE - enable receive and error interrupts

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_ERINT_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-889. MSCAN_ERINT_ENABLE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Error or receive interrupts to be enabled. Any combination of the following flags may be specified: MSCAN_WAKEUP_INT ... wake-up interrupt MSCAN_STATCHNG_INT ... status change interrupt MSCAN_OVERRUN_INT ... receiver overrun interrupt MSCAN_RXFULL_INT ... receiver full interrupt

Description: The *MSCAN_ERINT_ENABLE* ioctl command enables the selected MSCAN interrupts. Only the error or receiver interrupts can be enabled with this command. Also see the MSCAN_TINT_ENABLE ioctl command for a description how to enable transmitter-related interrupts.

The status-change interrupt sources and conditions need to be further configured using the MSCAN_ERINT_SET_RSTATE_MODE and MSCAN_ERINT_SET_TSTATE_MODE ioctl commands.

This command sets some or all of the WUPIE, CSCIE, OVRIE and RXFIE flags in the CANRIER register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_ERINT_ENABLE* ioctl command is implemented as a macro.

Example 5-777. MSCAN_ERINT_ENABLE

```
ioctl(MSCAN, MSCAN_ERINT_ENABLE, MSCAN_RXFULL_INT |
     MSCAN_RXOVR_INT);
```

This code enables the receiver-full and receiver-overflow interrupts.

5.25.3.31 MSCAN_ERINT_DISABLE - disable receive and error interrupts

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_ERINT_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-890. MSCAN_ERINT_DISABLE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Error or receive interrupts to be disabled. Any combination of the following flags may be specified: MSCAN_WAKEUP_INT ... wake-up interrupt MSCAN_STATCHNG_INT ... status change interrupt MSCAN_OVERRUN_INT ... receiver overrun interrupt MSCAN_RXFULL_INT ... receiver full interrupt

Description: The *MSCAN_ERINT_DISABLE* ioctl command disables the selected MSCAN interrupts. Only the error or receiver interrupts can be disabled with this command. Also see the MSCAN_TINT_DISABLE ioctl command for a description how to disable transmitter-related interrupts.

This command clears some or all of the WUPIE, CSCIE, OVRIE and RXFIE flags in the CANRIER register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_ERINT_DISABLE* ioctl command is implemented as a macro.

Example 5-778. MSCAN_ERINT_DISABLE

```
ioctl(MSCAN, MSCAN_ERINT_DISABLE, MSCAN_WAKEUP_INT);
```

This code disables the MSCAN wake-up interrupt.

5.25.3.32 MSCAN_ERINT_SET_RSTATE_MODE - configure receiver status-changed interrupt source

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_ERINT_SET_RSTATE_MODE,
     UWord16 param);
```

Arguments:

Table 5-891. MSCAN_ERINT_SET_RSTATE_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Select one of the status-change interrupt modes for the receiver operation: MSCAN_STAT_VOID ... do not generate the status-change interrupt for receiver status changes MSCAN_STAT_BUSOFF ... generate the status-change interrupt only if receiver enters or leaves the bus-off state (error counter > 255) MSCAN_STAT_ERR ... generate the status-change interrupt if receiver enters or leaves the RXERR or the bus-off state (error counter > 127) MSCAN_STAT_ALL ... generate the status-change interrupt on any receiver state change (error counter > 96)

Description: The *MSCAN_ERINT_SET_RSTATE_MODE* ioctl command sets up the receiver condition for generating the status-changed interrupt. There are four severity levels of receiver state defined by means of the receiver error counter value.

This command sets the RSTATE0 and RSTATE1 flags in the CANRIER register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_ERINT_SET_RSTATE_MODE* ioctl command is implemented as a macro.

Example 5-779. MSCAN_ERINT_SET_RSTATE_MODE

```
ioctl(MSCAN, MSCAN_ERINT_SET_RSTATE_MODE, MSCAN_STAT_BUSOFF);
```

This code enables the receiver status-change interrupt to be generated when receiver goes into the bus-off state.

5.25.3.33 MSCAN_ERINT_SET_TSTATE_MODE - configure transmitter status-changed interrupt source

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_ERINT_SET_TSTATE_MODE,
     UWord16 param);
```

Arguments:

Table 5-892. MSCAN_ERINT_SET_TSTATE_MODE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Select one of the status-change interrupt modes for the transmitter operation: MSCAN_STAT_VOID ... do not generate the status-change interrupt for transmitter status changes MSCAN_STAT_BUSOFF ... generate the status-change interrupt only if transmitter enters or leaves the bus-off state (error counter > 255) MSCAN_STAT_ERR ... generate the status-change interrupt if transmitter enters or leaves the RXERR or the bus-off state (error counter > 127) MSCAN_STAT_ALL ... generate the status-change interrupt on any transmitter state change (error counter > 96)

Description: The *MSCAN_ERINT_SET_TSTATE_MODE* ioctl command sets up the transmitter condition for generating the status-changed interrupt. There are four severity levels of transmitter state defined by means of the transmitter error counter value.

This command sets the TSTATE0 and TSTATE1 flags in the CANRIER register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_ERINT_SET_TSTATE_MODE* ioctl command is implemented as a macro.

Example 5-780. MSCAN_ERINT_SET_TSTATE_MODE

```
ioctl(MSCAN, MSCAN_ERINT_SET_TSTATE_MODE, MSCAN_STAT_BUSOFF);
```

This code enables the transmitter status-change interrupt to be generated when transmitter goes into the bus-off state.

5.25.3.34 MSCAN_TINT_ENABLE - enable transmitter interrupts

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_TINT_ENABLE,
     UWord16 param);
```

Arguments:

Table 5-893. MSCAN_TINT_ENABLE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Transmit buffer interrupts to be enabled. Any combination of the following flags may be specified: MSCAN_TXEMPTY0_INT | MSCAN_TXEMPTY1_INT | MSCAN_TXEMPTY2_INT MSCAN_TXEMPTY_ALL_INTS ... all flags combined together

Description: The *MSCAN_TINT_ENABLE* ioctl command enables the selected MSCAN interrupts. Only the transmitter buffer empty interrupts can be enabled with this command. Also see the MSCAN_ERINT_ENABLE ioctl command for a description how to enable the receiver- or error-related interrupts.

The bit-mapping of the MSCAN_TXEMPTY*x*_INT flags is the same as mapping of the MSCAN_TXBUFFER*x* flags, so a value returned by the MSCAN_SELECT_TXBUFF ioctl command or by the MSCAN_SELECT_NEXT_TXBUFF ioctl command may be used as a parameter for the MSCAN_TINT_ENABLE ioctl command (provided the value is non-zero).

This command sets some or all of the TXEIE0, TXEIE1 and TXEIE2 bits in the CANTCR (newly CANTIER) register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_TINT_ENABLE* ioctl command is implemented as a macro.

Example 5-781. MSCAN_TINT_ENABLE

```
ioctl(MSCAN, MSCAN_TINT_ENABLE, MSCAN_TXEMPTY0_INT);
```

This code enables the transmitter interrupt to be generated when transmitter buffer 0 is transmitted.

5.25.3.35 MSCAN_TINT_DISABLE - disable transmitter interrupts

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_TINT_DISABLE,
     UWord16 param);
```

Arguments:

Table 5-894. MSCAN_TINT_DISABLE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Transmit buffer interrupts to be disabled. Any combination of the following flags may be specified: MSCAN_TXEMPTY0_INT | MSCAN_TXEMPTY1_INT | MSCAN_TXEMPTY2_INT MSCAN_TXEMPTY_ALL_INTS ... all flags combined together

Description: The *MSCAN_TINT_DISABLE* ioctl command disables the selected MSCAN interrupts. Only the transmitter buffer empty interrupts can be disabled with this command. Also see the MSCAN_ERINT_DISABLE ioctl command for a description how to disable the receiver- or error-related interrupts.

This command clears some or all of the TXEIE0, TXEIE1 and TXEIE2 bits in the CANTCR (newly CANTIER) register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_TINT_DISABLE* ioctl command is implemented as a macro.

Example 5-782. MSCAN_TINT_DISABLE

```
ioctl(MSCAN, MSCAN_TINT_DISABLE, MSCAN_TXEMPTY_ALL_INTS);
```

This code disables all the transmitter-empty interrupts.

5.25.3.36 MSCAN_GET_ENABLED_TINT - get mask of enabled transmitter interrupts

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_GET_ENABLED_TINT,
      NULL);
```

Arguments:

Table 5-895. MSCAN_GET_ENABLED_TINT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_GET_ENABLED_TINT* ioctl command returns the mask of all Transmit Buffer Empty interrupts which are currently enabled. Typically, a value returned by this command is logically AND-ed with the value returned by the MSCAN_READ_TINT_FLAGS ioctl command at the beginning of the MSCAN transmitter interrupt handler. The result value can then be used to determine which transmit buffer needs servicing.

This command returns value of the CANTCR (newly CANTIER) register.

Returns: UWord16 register value.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_GET_ENABLED_TINT* ioctl command is implemented as a macro.

Example 5-783. MSCAN_GET_ENABLED_TINT

```
/* what tx buffers are empty and needs servicing? */
UWord16 tint = ioctl(MSCAN, MSCAN_READ_TINT_FLAGS, NULL);
tint &= ioctl(MSCAN, MSCAN_GET_ENABLED_TINT, NULL);

if(tint & MSCAN_TXBUFFER0)
      // transmit buffer 0 was recently transmitted
if(tint & MSCAN_TXBUFFER1)
      // transmit buffer 1 was recently transmitted
if(tint & MSCAN_TXBUFFER2)
      // transmit buffer 2 was recently transmitted

ioctl(MSCAN, MSCAN_INT_DISABLE, tint);
```

This code (in the MSCAN transmit interrupt handler) checks what transmit buffers were recently emptied. Such an approach requires an appropriate transmitter-empty interrupt is enabled always when a buffer is transmitted in the application (see MSCAN_TRANSMIT ioctl command).

5.25.3.37 MSCAN_READ_ERINT_FLAGS - get error and receiver interrupts pending

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_READ_ERINT_FLAGS,
      NULL);
```

Arguments:

Table 5-896. MSCAN_READ_ERINT_FLAGS ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_READ_ERINT_FLAGS* ioctl command returns a value of the Error and Receive interrupt flags. This command can be used to poll the state of the error and receiver interrupts. The value returned by this command may be used as a parameter to the MSCAN_CLEAR_ERINT_FLAGS ioctl command to acknowledge and clear the polled interrupt sources.

This command may also be used for read-clear sequence within the interrupt service routine, if this routine is used to handle both the error and receive interrupts.

Note that this command is *not* suitable for read-clear sequence within the interrupt service routine when two different service routines exist for the error and receive interrupts. If this is the case, use the MSCAN_READ_EINT_FLAGS and MSCAN_CLEAR_EINT_FLAGS ioctl commands for the error interrupt and the MSCAN_CLEAR_RINT_FLAGS ioctl command for the receiver interrupt.

This command returns value of the WUPIF, CSCIF, OVRIF and RXF bits of the CANRFLG register.

Returns: UWord16 register value.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_READ_ERINT_FLAGS* ioctl command is implemented as a macro.

Example 5-784. MSCAN_READ_ERINT_FLAGS

```
UWord16 poll;

poll = ioctl(MSCAN, MSCAN_READ_ERINT_FLAGS, NULL);

if(poll & MSCAN_RXFULL_INT)
     // receiver is full
// ... test other error and receive interrupt flags in the "poll"

ioctl(MSCAN, MSCAN_CLEAR_ERINT_FLAGS, poll);
```

This code polls the receiver-full interrupt flag.

5.25.3.38 MSCAN_READ_EINT_FLAGS - get error interrupts pending

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_READ_EINT_FLAGS,
      NULL);
```

Arguments:

Table 5-897. MSCAN_READ_EINT_FLAGS ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_READ_EINT_FLAGS* ioctl command returns the Error interrupt flags. This command is typically used in the MSCAN error interrupt service routine to retrieve the error interrupt events to be handled. The MSCAN_CLEAR_EINT_FLAGS ioctl command can then be used to clear such events by using the MSCAN_READ_EINT_FLAGS return value as a parameter.

This command returns value of the CSCIF, OVRIF bits of the CANRFLG register.

Returns: UWord16 register value.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_READ_EINT_FLAGS* ioctl command is implemented as a macro.

Example 5-785. MSCAN_READ_EINT_FLAGS

```
UWord16 isr;

isr = ioctl(MSCAN, MSCAN_READ_EINT_FLAGS, NULL);

if(isr & MSCAN_STATCHNG_INT)
      // status changed
if(isr & MSCAN_OVERRUN_INT)
      // status changed

ioctl(MSCAN, MSCAN_CLEAR_EINT_FLAGS, isr);
```

This code retrieves, processes and clears the error interrupt events within the error interrupt service routine.

5.25.3.39 MSCAN_READ_TINT_FLAGS - get transmitter interrupts pending

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_READ_TINT_FLAGS,
      NULL);
```

Arguments:

Table 5-898. MSCAN_READ_TINT_FLAGS ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_READ_TINT_FLAGS* ioctl command returns the Transmit Buffer Empty interrupt flags, one flag for each transmit buffer. This command can be used to determine what transmit buffers are empty.

When used in the transmitter interrupt service routine, the return value may need to be logically AND-ed with a value returned by the MSCAN_GET_ENABLED_TINT ioctl command so only the buffers which really caused the interrupt invocation are identified.

There is no way to clear the transmit buffer empty interrupt other than filling the buffer and submitting it for transmission using the MSCAN_TRANSMIT ioctl command. When there are no more data to be transmitted, the transmit buffer empty interrupt has to be disabled by using the MSCAN_TINT_DISABLE ioctl command.

This command returns value of the TXE0, TXE1 and TXE2 bits of the CANTFLG register.

Returns: UWord16 register value.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_READ_TINT_FLAGS* ioctl command is implemented as a macro.

Example 5-786. MSCAN_READ_TINT_FLAGS

```
/* what tx buffers are empty and needs servicing? */
UWord16 tint = ioctl(MSCAN, MSCAN_READ_TINT_FLAGS, NULL);
tint &= ioctl(MSCAN, MSCAN_GET_ENABLED_TINT, NULL);

if(tint & MSCAN_TXBUFFER0)
      // transmit buffer 0 was recently transmitted
if(tint & MSCAN_TXBUFFER1)
      // transmit buffer 1 was recently transmitted
if(tint & MSCAN_TXBUFFER2)
      // transmit buffer 2 was recently transmitted

ioctl(MSCAN, MSCAN_INT_DISABLE, tint);
```

This code (in the MSCAN transmit interrupt handler) checks what transmit buffers were recently emptied. Such an approach requires to enable always the appropriate transmitter-empty interrupt when a buffer is transmitted in the application (see the MSCAN_TRANSMIT ioctl command).

5.25.3.40 MSCAN_CLEAR_ERINT_FLAGS - clear error and receiver interrupts flags

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_CLEAR_ERINT_FLAGS,
     UWord16 param);
```

Arguments:

Table 5-899. MSCAN_CLEAR_ERINT_FLAGS ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Error or receive interrupt flags to be cleared. Any combination of the following flags may be specified: MSCAN_WAKEUP_INT ... wake-up interrupt MSCAN_STATCHNG_INT ... status change interrupt MSCAN_OVERRUN_INT ... receiver overrun interrupt MSCAN_RXFULL_INT ... receiver full interrupt

Description: The *MSCAN_CLEAR_ERINT_FLAGS* ioctl command clears the Error and Receive interrupt flags as specified in the parameter value.

See description of the MSCAN_READ_ERINT_FLAGS ioctl command for more details about identifying and acknowledging the error and receiver interrupt sources.

This command writes the parameter value to the WUPIF, CSCIF, OVRIF and RXF bits of the CANRFLG register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_CLEAR_ERINT_FLAGS* ioctl command is implemented as a macro.

Example 5-787. MSCAN_CLEAR_ERINT_FLAGS

```
UWord16 poll;

poll = ioctl(MSCAN, MSCAN_READ_ERINT_FLAGS, NULL);

if(poll & MSCAN_RXFULL_INT)
     // receiver is full
// ... test other error and receive interrupt flags in the "poll"

ioctl(MSCAN, MSCAN_CLEAR_ERINT_FLAGS, poll);
```

This code polls the receiver-full interrupt flag.

5.25.3.41 MSCAN_CLEAR_EINT_FLAGS - clear error interrupt flags

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_CLEAR_EINT_FLAGS,
      UWord16 param);
```

Arguments:

Table 5-900. MSCAN_CLEAR_EINT_FLAGS ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Error interrupt flags to be cleared. Any combination of the following flags may be specified: MSCAN_STATCHNG_INT ... status change interrupt MSCAN_OVERRUN_INT ... receiver overrun interrupt

Description: The *MSCAN_CLEAR_EINT_FLAGS* ioctl command clears the Error interrupt flags as specified in the parameter value.

See description of the MSCAN_READ_EINT_FLAGS ioctl command for more details about identifying and acknowledging the error interrupt sources.

This command writes the parameter value to the CSCIF and OVRIF bits of the CANRFLG register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_CLEAR_EINT_FLAGS* ioctl command is implemented as a macro.

Example 5-788. MSCAN_CLEAR_EINT_FLAGS

```
UWord16 isr;

isr = ioctl(MSCAN, MSCAN_READ_EINT_FLAGS, NULL);

if(isr & MSCAN_STATCHNG_INT)
      // status changed
if(isr & MSCAN_OVERRUN_INT)
      // status changed

ioctl(MSCAN, MSCAN_CLEAR_EINT_FLAGS, isr);
```

This code retrieves, processes and clears the error interrupt events within the error interrupt service routine.

5.25.3.42 MSCAN_CLEAR_RINT_FLAG - clear receive interrupt flag

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_CLEAR_RINT_FLAG,
     NULL);
```

Arguments:

Table 5-901. MSCAN_CLEAR_RINT_FLAG ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_CLEAR_RINT_FLAG* ioctl command clears the Receive Buffer Full interrupt flag. Typically, this command is used in the receiver interrupt service routine to clear and acknowledge the interrupt.

This command writes one to the RXF bit of the CANRFLG register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_CLEAR_RINT_FLAG* ioctl command is implemented as a macro.

Example 5-789. MSCAN_CLEAR_RINT_FLAG

```
ioctl(MSCAN, MSCAN_CLEAR_RINT_FLAG, NULL);
```

This code clears the receive buffer full interrupt flag.

5.25.3.43 MSCAN_CLEAR_WINT_FLAG - clear wake-up interrupt flag

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_CLEAR_WINT_FLAG,
     NULL);
```

Arguments:

Table 5-902. MSCAN_CLEAR_WINT_FLAG ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_CLEAR_WINT_FLAG* ioctl command clears the Wake-up interrupt flag. Typically, this command is used in the CAN wake-up interrupt service routine to clear and acknowledge the interrupt.

This command writes one to the WUPIF bit of the CANRFLG register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_CLEAR_WINT_FLAG* ioctl command is implemented as a macro.

Example 5-790. MSCAN_CLEAR_WINT_FLAG

```
ioctl(MSCAN, MSCAN_CLEAR_WINT_FLAG, NULL);
```

This code clears the CAN wake-up interrupt flag.

5.25.3.44 MSCAN_SELECT_TXBUFF - select given transmit buffer

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_SELECT_TXBUFF,
      UWord16 param);
```

Arguments:

Table 5-903. MSCAN_SELECT_TXBUFF ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Identifier of one or more transmit buffers. Any combination of the following flags may be specified: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2 MSCAN_ANY_TXBUFFER ... combination of all TXBUFFER flags

Description: The *MSCAN_SELECT_TXBUFF* ioctl command looks for an empty transmit buffer(s) from the ones specified by a parameter value. If at least one empty buffer is found, it is selected (mapped) to the MSCAN peripheral space and made available for the CPU access. A flag identifying such a “winning” buffer is then returned to the caller.

The zero value is returned when no of the specified transmit buffers is empty. In this case the buffer mapping is not changed.

If the MSCAN_ANY_BUFFER value is used as a parameter, this command behavior is the same as of the MSCAN_SELECT_NEXT_TXBUFF ioctl command.

This command writes the parameter value AND-ed with a value of the CANTFLG register to the CANTBSEL register.

Returns: A value of the CANTBSEL register which reflects the single TXBUFFER selected.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_SELECT_TXBUFF* ioctl command is implemented as a macro.

Example 5-791. MSCAN_SELECT_TXBUFF

```
if(ioctl(MSCAN, MSCAN_SELECT_TXBUFF, MSCAN_TXBUFFER0))
{
      // now fill the TX buffer registers

      // ....
      ioctl(MSCAN, MSCAN_TRANSMIT, MSCAN_TXBUFFER0);
}
```

This code tries to select the transmit buffer 0 into the MSCAN peripheral space and if successful it fills it and submits for a transmission. The “if” condition is not satisfied when the transmit buffer 0 is not empty (thus can not be accessed).

5.25.3.45 MSCAN_SELECT_NEXT_TXBUFF- select a free transmit buffer

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_SELECT_NEXT_TXBUFF,
      NULL);
```

Arguments:

Table 5-904. MSCAN_SELECT_NEXT_TXBUFF ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_SELECT_NEXT_TXBUFF* ioctl command looks for any empty transmit buffer and, if such is found, it selects it (maps it) to the MSCAN peripheral space. A flag identifying such a "winning" buffer is returned to the caller.

The zero value is returned when there is no transmit buffer empty. In this case the buffer mapping is not changed.

This command writes the value of the CANTFLG register into the CANTBSEL register.

Returns: A value of the CANTBSEL register which reflects the single TXBUFFER selected.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_SELECT_NEXT_TXBUFF* ioctl command is implemented as a macro.

Example 5-792. MSCAN_SELECT_NEXT_TXBUFF

```
// look for next empty buffer
UWord16 txbuff;
txbuff = ioctl(MSCAN, MSCAN_SELECT_NEXT_TXBUFF, NULL);

if(txbuff)
{
      // now fill the TX buffer registers
      // ....

      // and submit it for a transmission
      ioctl(MSCAN, MSCAN_TRANSMIT, txbuff);

      // raise an interrupt after this buffer is sent
      ioctl(MSCAN, MSCAN_TINT_ENABLE, txbuff);
}
```

This code selects (maps) one of empty transmit buffers to the MSCAN peripheral space, fills it and submits it for a transmission. An interrupt for the selected transmit buffer is then also enabled (see more details at MSCAN_READ_TINT_FLAGS).

The "txbuff" variable is set to zero when there is no empty buffer available.

5.25.3.46 MSCAN_TRANSMIT - select given transmit buffer

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_TRANSMIT,
      UWord16 param);
```

Arguments:

Table 5-905. MSCAN_TRANSMIT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Identifier of one or more transmit buffers. Any combination of the following flags may be specified: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2

Description: The *MSCAN_TRANSMIT* ioctl command submits the transmit buffer(s) selected by the parameter value for a transmission onto the CAN bus. Once submitted, the buffers are marked as full and should not be accessed by CPU. Submitted buffer(s) go empty either if transmission is aborted or successfully completed.

This command writes the parameter value into the CANTFLG register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_TRANSMIT* ioctl command is implemented as a macro.

Example 5-793. MSCAN_TRANSMIT

```
// look for next empty buffer
UWord16 txbuff;
txbuff = ioctl(MSCAN, MSCAN_SELECT_NEXT_TXBUFF, NULL);

if(txbuff)
{

      // now fill the TX buffer registers
      // ....

      // and submit it for a transmission
      ioctl(MSCAN, MSCAN_TRANSMIT, txbuff);

      // raise an interrupt after this buffer is sent
      ioctl(MSCAN, MSCAN_TINT_ENABLE, txbuff);
}
```

This code selects (maps) one of empty transmit buffers to the MSCAN peripheral space, fills it and submits it for a transmission. An interrupt for the selected transmit buffer is then also enabled (see more details at MSCAN_READ_TINT_FLAGS).

5.25.3.47 MSCAN_ABORT_TRANSMIT - abort buffer transmission

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_ABORT_TRANSMIT,
     UWord16 param);
```

Arguments:

Table 5-906. MSCAN_ABORT_TRANSMIT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.
param	in	Identifier of one or more transmit buffers. Any combination of the following flags may be specified: MSCAN_TXBUFFER0 | MSCAN_TXBUFFER1 | MSCAN_TXBUFFER2

Description: The *MSCAN_ABORT_TRANSMIT* ioctl command cancels (aborts) the transmission of the selected buffers. The cancellation is successful if the message is not already in transmission, or if the transmission is not successful (lost arbitration or error). When a message is aborted, the associated buffer is marked as empty and is reported in the abort acknowledge register. See also the MSCAN_READ_ABORT_ACK ioctl command.

This command writes the parameter value to the CANTARQ register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_ABORT_TRANSMIT* ioctl command is implemented as a macro.

Example 5-794. MSCAN_ABORT_TRANSMIT

```
ioctl(MSCAN, MSCAN_ABORT_TRANSMIT, MSCAN_TXBUFFER_0);
```

This code un-schedules the transmit buffer 0 from a transmission process.

5.25.3.48 MSCAN_READ_ABORT_ACK - get aborted buffers

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_READ_ABORT_ACK, NULL);
```

Arguments:

Table 5-907. MSCAN_READ_ABORT_ACK ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_READ_ABORT_ACK* ioctl command returns the transmission abort acknowledge flags for those buffers which were not transmitted due to a call of the MSCAN_ABORT_TRANSMIT ioctl command. The returned value may be tested for occurrence of MSCAN_TXBUFFER*x* flags.

The transmission abort acknowledge flag is cleared when the associated buffer is transmitted next time (using the MSCAN_TRANSMIT ioctl command).

Returns: A value of the CANTAAK register.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_READ_ABORT_ACK* ioctl command is implemented as a macro.

Example 5-795. MSCAN_READ_ABORT_ACK

```
if(ioctl(MSCAN, MSCAN_READ_ABORT_ACK, NULL) & MSCAN_TXBUFFER_0)
{
      // transmit buffer 0 transmission was successfully aborted
}
```

This code tests if transmission cancellation of buffer 0 was successful.

5.25.3.49 MSCAN_TEST_BUSOFF_HOLD - test if manual bus-off recovery is needed

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_TEST_BUSOFF_HOLD,
      NULL);
```

Arguments:

Table 5-908. MSCAN_TEST_BUSOFF_HOLD ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_TEST_BUSOFF_HOLD* ioctl command returns a non-zero value when the MSCAN module is stuck in the bus-off state. This may happen if manual bus-off recovery mode was configured for the MSCAN module. The bus-off state may be recovered using the MSCAN_RECOVER_BUSOFF_STATE ioctl command.

This command returns a value of the BOHOLD bit of the CANMISC register.

Returns: UWord16 value for zero/non-zero testing.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_TEST_BUSOFF_HOLD* ioctl command is implemented as a macro.

Example 5-796. MSCAN_TEST_BUSOFF_HOLD

```
if(ioctl(MSCAN, MSCAN_TEST_BUSOFF_HOLD, NULL))
{
      // bus-off state detected, do recover now
      ioctl(MSCAN, MSCAN_RECOVER_BUSOFF_STATE, NULL);
}
```

This code tests if module is in the bus-off state and recovers from this state.

5.25.3.50 MSCAN_RECOVER_BUSOFF_STATE - manual recovery from bus-off state

Call(s):

```
void ioctl(const int pModuleBase, MSCAN_RECOVER_BUSOFF_STATE,
     NULL);
```

Arguments:

Table 5-909. MSCAN_RECOVER_BUSOFF_STATE ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_RECOVER_BUSOFF_STATE* ioctl command requests a recovery from the bus-off state. This command needs to be used to recover from the bus-off state if manual bus-off recovery mode was configured for the MSCAN module and the bus-off state is detected using the MSCAN_TEST_BUSOFF_HOLD ioctl command.

This command writes one to the BOHOLD bit of the CANMISC register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_RECOVER_BUSOFF_STATE* ioctl command is implemented as a macro.

Example 5-797. MSCAN_RECOVER_BUSOFF_STATE

```
if(ioctl(MSCAN, MSCAN_RECOVER_BUSOFF_STATE, NULL))
{
     // bus-off state detected, do recover now
     ioctl(MSCAN, MSCAN_RECOVER_BUSOFF_STATE, NULL);
}
```

This code tests if module is in the bus-off state and recovers from this state.

5.25.3.51 MSCAN_GET_WINNING_ACC_FILTER - test identifier acceptance filter hit

Call(s):

```
UWord16 ioctl(const int pModuleBase,MSCAN_GET_WINNING_ACC_FILTER,
      NULL);
```

Arguments:

Table 5-910. MSCAN_GET_WINNING_ACC_FILTER ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_GET_WINNING_ACC_FILTER* ioctl command returns an index of the acceptance filter (0-7) which was hit during reception of a message currently available in the receive buffer.

This command returns a value of the IDHIT0, IDHIT1 and IDHIT2 bits of the CANIDAC register.

Returns: UWord16 index value in range 0-7.

Range Issues: None.

Special Issues: This command is applicable only on MC56F803x.

Design/Implementation: The *MSCAN_GET_WINNING_ACC_FILTER* ioctl command is implemented as a macro.

Example 5-798. MSCAN_GET_WINNING_ACC_FILTER

```
UWord16 idhit;

idhit = if(ioctl(MSCAN, MSCAN_GET_WINNING_ACC_FILTER, NULL);
```

This code reads the index of the identifier acceptance filter which enabled a reception of the currently-received message.

5.25.3.52 MSCAN_GET_RX_ERR_COUNT- read RX error counter

Call(s):

```
UWord16 ioctl(const int pModuleBase, MSCAN_GET_RX_ERR_COUNT,
      NULL);
```

Arguments:

Table 5-911. MSCAN_GET_RX_ERR_COUNT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_GET_RX_ERR_COUNT* ioctl command returns the 8-bit value of the MSCAN Receive Error Counter (CANRXERR). See the MSCAN documentation for a detailed description of error counter values.

Returns: Content of the CANRXERR register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCAN_GET_RX_ERR_COUNT* ioctl command is implemented as a macro.

Example 5-799. MSCAN_GET_RX_ERR_COUNT

```
UWord8 rxErr;

rxErr = ioctl(MSCAN, MSCAN_GET_RX_ERR_COUNT, NULL);
```

This code reads the value of the MSCAN receiver error counter.

5.25.3.53 MSCAN_GET_TX_ERR_COUNT - read TX error counter

Call(s):

```
UWord8 ioctl(const int pModuleBase, MSCAN_GET_TX_ERR_COUNT,
             NULL);
```

Arguments:

Table 5-912. MSCAN_GET_TX_ERR_COUNT ioctl call arguments

pModuleBase	in	The MSCAN module identifier. Use MSCAN.

Description: The *MSCAN_GET_TX_ERR_COUNT* ioctl command returns the 8-bit value of the MSCAN Transmit Error Counter Register (CANTXERR). See the MSCAN documentation for a detailed description of error counter values.

Returns: Content of the CANTXERR register.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCAN_GET_TX_ERR_COUNT* ioctl command is implemented as a macro.

Example 5-800. MSCAN_GET_TX_ERR_COUNT

```
UWord8 txErr;

txErr = ioctl(MSCAN, MSCAN_GET_TX_ERR_COUNT, NULL);
```

This code reads the value of the MSCAN transmitter error counter.

5.25.4 Message Buffer API Specification

This section describes the ioctl commands used to access the receive or transmit message buffer registers. The commands are prefixed with MSCANMB to notify user that a different module identifier is used.

- For the receive message buffer, use the MSCAN_RB identifier
- For the currently selected transmit buffer, use the MSCAN_TB identifier (see more details about buffer mapping in the MSCAN_SELECT_TXBUFF or MSCAN_SELECT_NEXT_TXBUFF ioctl command description).
- For a custom message buffer located in the memory (memory block copied-out of the receive buffer, or prepared to be copied into the transmit buffer), use the memory buffer address “casted” to *arch_sMSCAN_MB* pointer type.

5.25.4.1 MSCANMB_GET_ID - get message identifier

Call(s):

```
UWord32 ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_ID, NULL);
```

Arguments:

Table 5-913. MSCANMB_GET_ID ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_ID* ioctl command retrieves the CAN frame identifier of the message currently stored in the Message Buffer.

The returned Message Identifier is converted from a raw form of the MB structure so that it can be treated as a standard 32-bit number. The most significant bit (MSB) of the return value is set for the extended 29-bit identifiers. The MSB is cleared for the standard 11-bit identifiers. Several examples of possible return values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567
0x80000033   ... Extended identifier 0x33
```

Returns: CAN frame identifier value. The most significant bit identifies extended (MSB=1) or standard (MSB=0) CAN identifier. You can use the MSCAN_ID_EXT constant to test the MSB bit.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_ID* ioctl command is implemented as a function call.

Example 5-801. MSCANMB_GET_ID

```
UWord32 id;

id = ioctl(MSCAN_RB, MSCANMB_GET_ID, NULL);
```

This code retrieves the message ID of the receive buffer.

5.25.4.2 MSCANMB_GET_ID_RAW - get message identifier registers

Call(s):

```
UWord32 ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_ID_RAW, NULL);
```

Arguments:

Table 5-914. MSCANMB_GET_ID_RAW ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_ID_RAW* ioctl command returns the four IDR registers of MB as a single 32 bit word.

The returned value is in the raw format as it is stored in the MB registers so further conversions are needed to obtain a numeric representation of the message identifier.

Returns: CAN frame identifier value in raw (IDR registers) format.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_ID* ioctl command is implemented as a function call.

Example 5-802. MSCANMB_GET_ID_RAW

```
UWord32 idr;

idr = ioctl(MSCAN_RB, MSCANMB_GET_ID_RAW, NULL);
```

This code retrieves ID-raw value of a message received in the receive buffer.

5.25.4.3 MSCANMB_GET_LEN - get data length

Call(s):

```
UWord16 ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_LEN, NULL);
```

Arguments:

Table 5-915. MSCANMB_GET_LEN ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_LEN* ioctl command retrieves the length of the frame data part currently stored in the Message Buffer.

Returns: Length of the message stored in the message buffer (a UWord16 number in the range of 0...8)

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_LEN* ioctl command is implemented as a macro.

Example 5-803. MSCANMB_GET_LEN

```
UWord16 len;

len = ioctl(MSCAN_RB, MSCANMB_GET_LEN, NULL);
```

This code retrieves the length of the message in the receive buffer.

5.25.4.4 MSCANMB_GET_RTR - get remote transmit request flag

Call(s):

```
UWord16 ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_RTR, NULL);
```

Arguments:

Table 5-916. MSCANMB_GET_LEN ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_RTR* ioctl command returns non-zero if the RTR flag is set in the Message Buffer.

Returns: UWord16 for zero/non-zero testing. Non-zero returned when RTR is set in given MB.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_RTR* ioctl command is implemented as a macro.

Example 5-804. MSCANMB_GET_RTR

```
if(ioctl(MSCAN_RB, MSCANMB_GET_RTR, NULL))
{
      // handle remote-transmit-request
}
else
{
      // handle normal CAN frame
}
```

This code tests the status of the RTR bit of the message in the receive buffer.

5.25.4.5 MSCANMB_GET_DATAPTR - get pointer to frame data

Call(s):

```
UWord16* ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_DATAPTR,
           NULL);
```

Arguments:

Table 5-917. MSCANMB_GET_DATAPTR ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_DATAPTR* ioctl command returns the pointer to the data part of the message currently stored in the Message Buffer.

The returned pointer points to the eight-word data space in which up to eight message bytes are stored. Use the *MSCANMB_GET_LEN* ioctl command to determine the number of valid data bytes in the message.

Returns: Pointer to eight data words containing up to eight data bytes (each byte in a separate UWord16 value).

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_DATAPTR* ioctl command is implemented as a macro.

Example 5-805. MSCANMB_GET_DATAPTR

```
UWord16 len;
UWord16* pdata

len = ioctl(MSCAN_RB, MSCANMB_GET_LEN, NULL);
pdata = ioctl(MSCAN_RB, MSCANMB_GET_DATAPTR, NULL);
```

This code retrieves the length and pointer to data part of the frame in the receive buffer.

5.25.4.6 MSCANMB_GET_TIMESTAMP - get message time-stamp value

Call(s):

```
UWord16 ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_GET_TIMESTAMP,
          NULL);
```

Arguments:

Table 5-918. MSCANMB_GET_TIMESTAMP ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).

Description: The *MSCANMB_GET_TIMESTAMP* ioctl command returns the time-stamp value which was assigned to this message buffer. The time-stamp is a value of the free-running timer/ counter captured at the time the message was received or transmitted from/to the CAN bus. The time-stamp timer must be enabled in order to generate valid time-stamp values.

See MSCAN_TIMESTAMP_TIMER ioctl command for more details.

Returns: UWord16 time-stamp value.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_GET_TIMESTAMP* ioctl command is implemented as a macro.

Example 5-806. MSCANMB_GET_TIMESTAMP

```
UWord16 ts;

ts = ioctl(MSCAN_RB, MSCANMB_GET_TIMESTAMP, NULL);
```

This code retrieves the time-stamp value saved in the receive buffer.

5.25.4.7 MSCANMB_SET_ID - set message identifier (inline function)

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_ID,
           UWord32 param);
```

Arguments:

Table 5-919. MSCANMB_SET_ID ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	32bit Identifier value (constant). The most significant bit, when set, identifies the extended message identifier. Use an OR-combination of the ID numeric value and the following constants: MSCAN_ID_EXT ... when passing extended identifier MSCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *MSCANMB_SET_ID* ioctl command assigns the CAN message identifier to the specified Message Buffer, and clears or sets the Remote Transmit Request flag in the MB.

The identifier value passed to this ioctl command should contain information about whether it is specified in the standard 11-bit format or extended 29-bit format. The most significant bit (MSB) is reserved for this purpose. You can use the predefined MSCAN_ID_EXT constant OR-ed with the ID value being passed to the ioctl to set the MSB. Several examples of composing the identifier and mask values follow:

```
0x123        ... Standard identifier 0x123
0x22         ... Standard identifier 0x22
0x81234567   ... Extended identifier 0x1234567 (= 0x1234567 | MSCAN_ID_EXT)
0x80000033   ... Extended identifier 0x33 (= 0x33 | MSCAN_ID_EXT)
0xC0000033   ... Extended identifier 0x33 with RTR set (= 0x33 | MSCAN_ID_EXT |
                 MSCAN_ID_RTR)
```

By default, this command clears the RTR flag, which is stored in the IDR1 or IDR3 Message Buffer registers - depending on the message type. Use bit 30 of the ID passed to this command to set the RTR bit in the appropriate register. Bit 30 is defined as the MSCAN_ID_RTR constant in *mscan.h*. You can also use the *MSCANMB_SET_RTR* ioctl command to set or clear the RTR bit after the frame ID is already stored in the Message Buffer.

Returns: None.

Range Issues: The parameter contains the message ID for which the following range apply: The standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. The extended identifiers (MSB=1) are limited to 29 bits and the valid range is 0x8000000... 0x9fffffff.

Special Issues: This command uses the *MSCAN_Id2Idr()* macro to convert the passed value into the raw format suitable for writing into the IDR registers. The conversion process is not trivial and may produce quite lengthy code when passing a UWord32 variable as the parameter. Use this command for passing constant values *only* and use the *MSCANMB_SET_ID_V* ioctl command when passing variables.

Design/Implementation: The *MSCANMB_SET_ID* ioctl command is implemented as an inline function.

Example 5-807. MSCANMB_SET_ID

```
UWord16 tb;

tb = ioctl(MSCAN, MSCAN_SELECT_NEXT_TXBUFF, NULL);

if(tb)
{
      ioctl(MSCAN_TB, MSCANMB_SET_ID, 0x123 | MSCAN_ID_EXT |
            MSCAN_ID_RTR);
}
```

This code sets the message identifier of the selected transmit buffer to be the Extended ID 0x123. It also sets the RTR bit so the remote-transmit-request will be transmitted.

5.25.4.8 MSCANMB_SET_ID_V - set message identifier (function call)

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_ID_V,
           UWord32 param);
```

Arguments:

Table 5-920. MSCANMB_SET_ID_V ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	32bit Identifier value (variable). The most significant bit, when set, identifies the extended message identifier. Use an OR-combination of the ID numeric value and the following constants: MSCAN_ID_EXT ... when passing extended identifier MSCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *MSCANMB_SET_ID_V* ioctl command assigns the CAN message identifier to the specified Message Buffer. This command operates in the same way as the *MSCANMB_SET_ID* ioctl command, except that the *MSCANMB_SET_ID_V* ioctl command is implemented as a function call and it is thus more suitable for passing the 32-bit identifier stored in the caller's variable.

Returns: None.

Range Issues: The parameter contains the message id for which the following ranges apply: The standard identifiers (MSB=0) are limited to 11 bits so the valid range is 0...0x7ff. The extended identifiers (MSB=1) are limited to 29 bits and the valid range is 0x8000000... 0x9fffffff.

Special Issues: None.

Design/Implementation: The *MSCANMB_SET_ID_V* ioctl command is implemented as a function call. This function call saves the contents of the CPU registers, so it can be called from interrupt service routines.

Example 5-808. MSCANMB_SET_ID_V

```
UWord32 id = 0x123 | MSCAN_ID_EXT;

ioctl(MSCAN_TB, MSCANMB_SET_ID_V, id);
```

This code sets the message identifier of the currently-selected transmit buffer to be the Extended ID 0x123.

5.25.4.9 MSCANMB_SET_ID_RAW - set message identifier registers

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_ID_RAW,
           UWord32 param);
```

Arguments:

Table 5-921. MSCANMB_SET_ID_RAW ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	32bit ID-raw value. MSCAN_ID_EXT ... when passing extended identifier MSCAN_ID_RTR ... before sending a Remote Transmit Request frame

Description: The *MSCANMB_SET_ID_RAW* ioctl command assigns the CAN message identifier to the specified Message Buffer.

This command behaves in a similar manner as the *MSCANMB_SET_ID* ioctl command, except that the passed identifier value must already be in the raw format suitable for writing into the IDR registers of the Message Buffer structure.

Returns: None.

Range Issues: See the MSCAN module documentation for the details about the Message Buffer identifier registers.

Special Issues: None.

Design/Implementation: The *MSCANMB_SET_ID_RAW* ioctl command is implemented as a macro.

Example 5-809. MSCANMB_SET_ID_RAW

```
ioctl(MSCAN_TB, MSCANMB_SET_ID_RAW, 0x11 << 21);
```

This code sets the message identifier of the currently-selected transmit buffer to 0x11.

5.25.4.10 MSCANMB_SET_RTR - set remote transmit request flag

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_RTR, param);
```

Arguments:

Table 5-922. MSCANMB_SET_RTR ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	Parameter to select the desired action. Use one of these predefined constants: MSCAN_ON - sets RTR MSCAN_OFF - clears RTR

Description: The *MSCANMB_SET_RTR* ioctl command sets or clears the Remote Transmit Request (RTR) bit in the message identifier value of the specified Message Buffer.

As an alternative to this command, you can set the RTR bit directly, when setting the Message Buffer ID value, using the *MSCANMB_SET_ID* or *MSCANMB_SET_ID_V* ioctl commands. The RTR bit is also directly accessed when writing the raw ID value using the *MSCAN_SET_ID_RAW* command.

The MSCANMB_SET_RTR ioctl command may be used after the ID is already set in the Message Buffer, to set or clear the RTR flag only.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: The *MSCANMB_SET_RTR* ioctl command is implemented as a macro.

Example 5-810. MSCANMB_SET_RTR

```
ioctl(MSCAN_TB, MSCANMB_SET_LEN, 0);
ioctl(MSCAN_TB, MSCANMB_SET_ID, 0x11);
ioctl(MSCAN_TB, MSCANMB_SET_RTR, MSCAN_ON);
```

This code prepares the transmission of the Remote Transmit Request of the ID 0x11 from the currently-selected transmit buffer.

5.25.4.11 MSCANMB_SET_LEN - set message length

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_LEN,
           UWord16 param);
```

Arguments:

Table 5-923. MSCANMB_SET_LEN ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	The length of the message to be set to the Message Buffer.

Description: The *MSCANMB_SET_LEN* ioctl command sets the message length field of the specified Message Buffer structure. Typically the length is set before the message is to be transmitted onto the CAN bus.

Returns: None.

Range Issues: The parameter must be in the range from 0 to 8.

Special Issues: None.

Design/Implementation: The *MSCANMB_SET_LEN* ioctl command is implemented as a macro.

Example 5-811. MSCANMB_SET_LEN

```
ioctl(MSCAN_TB, MSCANMB_SET_LEN, 4);
```

This code sets the message length of the currently-selected transmit buffer to four.

5.25.4.12 MSCANMB_SET_TBP - set transmit buffer priority

Call(s):

```
void ioctl(arch_sMSCAN_MB *pMBBase, MSCANMB_SET_TBP,
           UWord16 param);
```

Arguments:

Table 5-924. MSCANMB_SET_TBP ioctl call arguments

pMBBase	in	The MSCAN Message Buffer module identifier: MSCAN_RB ... receive buffer MSCAN_TB ... transmit buffer currently mapped into the peripheral space (see MSCAN_SELECT_TXBIUFF or MSCAN_SELECT_NEXT_TXBUFF).
param	in	The local priority to be set to the associated Message Buffer.

Description: The MSCANMB_SET_TBP ioctl command sets the local priority of the associated Message Buffer structure. The local priority is used for the internal prioritization process of the CAN and is defined to be highest for the smallest binary number.

Returns: None.

Range Issues: The parameter must be in the range from 0 to 255.

Special Issues: None.

Design/Implementation: The *MSCANMB_SET_TBP* ioctl command is implemented as a macro.

Example 5-812. MSCANMB_SET_TBP

```
ioctl(MSCAN_TB, MSCANMB_SET_TBP, 2);
```

This code sets the local priority of the currently-selected transmit buffer to the value of two.

5.25.5 MSCAN Driver Application

There is one sample application demonstrating how the MSCAN module can be used in a user application. The application can be found in the ***{DSP56800E_Quick_Start Source}\..\ sample_applications\EVM\mscan_demo*** directory**.** The only supported EVM board for the MSCAN sample application is MC56F8037EVM.

5.26 RTC Driver

This section describes the API of MC56F800x Real-Timer Counter (RTC) on-chip module. The functionality of the RTC module itself is described in the **56F800x Peripheral Reference Manual**.

5.26.1 Introduction

The MC56F800x devices have one Real-Time Counter module. The RTC module contains a 8-bit up counter, a modulo register, and a control register. The modulo and control registers are read/writable. The counter is read only. The modulo register is loaded with a value to count to and the pre-scaler is set to determine the counting rate. When enabled, the counter counts up to the modulo value and sets a flag (and an interrupt request if enabled), then it resets to $00, and resumes counting.

This section describes the RTC driver software, providing the low level software layer interfacing hardware with software.

5.26.2 Quick Reference

This section is intended as a source of quick access information, while the details are discussed in Section 5.26.3.

Table 5-925. RTC Module Base Address

Module base address of / for	MC56F800x	MC56F801x	MC56F802x/3x	MC56F83xx
RTC (RTC_BASE)	0xF320	N/A	N/A	N/A

5.26.2.1 API Definition

The following header files are needed in order to use the RTC device driver:

Required Header File(s):

```
#include "qs.h"
#include "rtc.h"
```

The following information may be found in the header file *rtc.h*.

Public Data Structure(s):

None

5.26.2.2 Configuration Items

This section summarizes the symbols used in macro definitions for the static RTC module configuration by the driver initialization routine. Configuration symbols are intended for the application (project) specific configuration file *appconfig.h*.

Table 5-926. RTC Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
RTC_RTCSC_INIT	UWord16	The initial value of the RTC Status and Control Register
RTC_RTCMOD_INIT	UWord16	Initial value of the RTC Modulo Register

5.26.2.3 API Specification

This section specifies the exact usage for each API function.

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Note: *inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

ioctl call(s):

The *ioctl* call is generally represented by one of the following forms:

```
UWord16 ioctl(const int *pModuleBase, void cmd, UWord16 param);
UWord16 ioctl(const int *pModuleBase, void cmd, void* pParam);
```

Description: The *ioctl* call "changes" the RTC device modes or accesses the RTC register(s). The third ioctl parameter is either a value or a pointer, depending on the type of cmd.

Arguments:

Table 5-927. RTC Driver Arguments - ioctl

pModuleBase	in	RTC module identifier. Use RTC.
cmd	in	Commands found in *rtc.h* which are used to modify the RTC module status and control registers. See Table 5-928.
param, pParam	in, inout	Used to pass the relevant data to ioctl function call.

Items Separators Convention:

```
/  only one of the specified items is allowed
|  consolidation of items is allowed ( item1 | item2 | item3 )
&  intersection of items is allowed ( item1 & item2 & item3 )
```

Table 5-928. ioctl commands

cmd	param	Return	Description
RTC_INIT	NULL	None	Initializes RTC module by data from the configuration file (appconfig.h).
RTC_SET_CLOCK_SOURCE_ PRESCALER	This command accepts always one of the pre-defined values generally formatted as: RTC_CLOCK_*source_DIV _prescaler* where *source* is clock source (e.g. 1KHZ, EXT, INT). The *prescaler* is a divide-by value for the clock source. For example: RTC_CLOCK_*INT_DIV_1* RTC_CLOCK_*INT_DIV_ OFF*	None	Sets RTC counter prescaler and clock source.
RTC_SET_CLOCK_ PRESCALER	RTC_CLOCK_RTCLKS_0 _x/ (x=OFF/1/2/4/8/10/16/32/64/100/128/256/500/512/1000/1024) RTC_CLOCK_RTCLKS_0 _x/ (x=OFF/1000/1024/2000/2048/4096/5000/8192/10000/16384/20000/32768/50000/65536/100000/200000)	None	Sets RTC counter prescaler.
RTC_CLOCK_SOURCE	RTC_CLOCK_1KHZ/ RTC_CLOCK_EXT/ RTC_CLOCK_INT/ RTC_CLOCK_INT_2	None	Selects RTC prescaler clock sourcer.
RTC_SET_INTERRUPT_ ENABLE	NULL	None	Enables RTC interrupt
RTC_SET_INTERRUPT_ DISABLE	NULL	None	Disables RTC interrupt

Table 5-928. ioctl commands (Continued)

cmd	param	Return	Description
RTC_TEST_INTERRUPT_ FLAG	NULL	None	Tests RTC interrupt flag
RTC_CLEAR_INTERRUPT_ FLAG	NULL	None	Clears RTC interrupt flag
RTC_WRITE_MODULO_REG	UWord16	None	Sets RTC modulo value.
RTC_READ_MODULO_REG	NULL	UWord16	Reads RTC modulo value.
RTC_READ_COUNTER_REG	NULL	UWord16	Reads RTC immediate counter value.

5.26.3 Detailed API Specification

The detailed functionality of all ioctl commands is explained in this section. The code examples illustrate the usage of the ioctl commands.

5.26.3.1 RTC_INIT - initialize timer module

Call(s):

```
void ioctl(const int *pModuleBase, RTC_INIT, NULL);
```

Arguments:

Table 5-929. RTC_INIT ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_INIT* ioctl command executes the initialization function, which initializes all configured RTC registers by the values defined in *appconfig.h*. It is intended for static configuration of the RTC module after power-up.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_INIT* ioctl command is implemented as a function call. Only necessary RTC registers, which are defined in *appconfig.h*, are initialized. The execution time depends on the number of defined registers.

Example 5-813. RTC_INIT

```
ioctl(RTC, RTC_INIT, NULL);
```

This code initializes the RTC modules by the values defined in appconfig.h. The appconfig.h file can be edited manually or generated by the Graphical Configuration Tool.

5.26.3.2 RTC_SET_CLOCK_SOURCE_PRESCALER - select prescaler source and counter prescaler

Call(s):

```
void ioctl(const int *pModuleBase,
           RTC_SET_CLOCK_SOURCE_PRESCALER, UWord16 param);
```

Arguments:

Table 5-930. RTC_SET_CLOCK_SOURCE_PRESCALER ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.
param	in	This command accepts always one of the pre-defined values generally formatted as: RTC_CLOCK_*source_DIV_prescaler* where *source* is clock source (e.g. 1KHZ, EXT, INT). The *prescaler* is a divide-by value for the clock source. For example: RTC_CLOCK_*INT_DIV_1* RTC_CLOCK_*INT_DIV_OFF* For a complete list see rtc.h.

Description: The *RTC_SET_CLOCK_SOURCE_PRESCALER* ioctl command sets the clock source and the prescaler counter. There is possible choose one of the 1KHZ/INT/EXT clock source. Value of the counter prescaler depends on selected clock source. See more details at the RTC_SET_CLOCK_SOURCE_PRESCALER command description.

This command writes directly into the Real-Time Clock Prescaler Select (RTCPS) bits and into the Real-Time Source Selects (RTCLKS) bits of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_SET_CLOCK_SOURCE_PRESCALER* command is implemented as a macro.

Example 5-814. RTC_SET_CLOCK_SOURCE_PRESCALER

```
ioctl(RTC, RTC_SET_CLOCK_SOURCE_PRESCALER,
      RTC_CLOCK_INT_DIV_1000);
```

This code resets RTC and sets internal clock source with 1000 prescaler.

5.26.3.3 RTC_SET_CLOCK_PRESCALER - select counter prescaler

Call(s):

```
void ioctl(const int *pModuleBase,RTC_SET_CLOCK_PRESCALER,
           UWord16 param);
```

Arguments:

Table 5-931. RTC_SET_CLOCK_PRESCALER ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.
param	in	This command accepts always one of the pre-defined values generally formatted as: RTC_CLOCK_*source_DIV_prescaler* where *source* is actual setting of clock source (e.g. RTCLKS0_0 or RTCLKS0_1). The *prescaler* is a divide-by value for the clock source. For example: RTC_CLOCK_*RTCLK0_1_DIV_1000* RTC_CLOCK_*RTCLK0_0_DIV_1* For a complete list see rtc.h.

Description: The *RTC_SET_CLOCK_PRESCALER* ioctl command sets the clock counter prescaler. Value of the counter prescaler depends on selected clock source. See more details at the RTC_SET_CLOCK_PRESCALER description.

This command writes directly into the Real-Time Prescaler Selects (RTCPS) bits of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_SET_CLOCK_PRESCALER* command is implemented as a macro.

Example 5-815. RTC_SET_CLOCK_PRESCALER

```
ioctl(RTC, RTC_SET_CLOCK_PRESCALER,RTC_CLOCK_RTCLK0_0_DIV_100);
```

This code resets RTC and sets clock source with 100 prescaler.

5.26.3.4 RTC_CLOCK_SOURCE - select the RTC clock source

Call(s):

```
void ioctl(const int *pModuleBase, RTC_CLOCK_SOURCE,
           UWord16 param);
```

Arguments:

Table 5-932. RTC_CLOCK_SOURCE ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.
param	in	Use one of following values: RTC_CLOCK_1KHZ ... 1KHz oscillator is used RTC_CLOCK_EXT ... External clock is used RTC_CLOCK_INT ... Internal clock is used RTC_CLOCK_INT_2 ... Internal clock is used with second divider

Description: The *RTC_CLOCK_SOURCE* ioctl command selects the RTC clock source.

This command writes directly into the Real-Time Source Selects (RTCLKS) bits of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_CLOCK_SOURCE* command is implemented as a macro.

Example 5-816. RTC_CLOCK_SOURCE

```
ioctl(RTC, RTC_CLOCK_SOURCE, RTC_CLOCK_EXT);
```

This code selsects external clock as the RTC clock source.

5.26.3.5 RTC_SET_INTERRUPT_ENABLE - enable RTC interrupt

Call(s):

```
void ioctl(const int *pModuleBase, RTC_SET_INTERRUPT_ENABLE,
           NULL);
```

Arguments:

Table 5-933. RTC_SET_INTERRUPT_ENABLE ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_SET_INTERRUPT_ENABLE* ioctl command enables the RTC interrupt to be generated. The RTC interrupt is generated when a RTC counter value wraps to zero after reaching the modulo value.

This command writes directly into the RTC Real-Time Interrupt Enable (RTIE) bit of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_SET_INTERRUPT_ENABLE* command is implemented as a macro.

Example 5-817. RTC_SET_INTERRUPT_ENABLE

```
ioctl(RTC, RTC_SET_INTERRUPT_ENABLE, NULL);
```

This code enables the RTC Real-Time interrupt.

5.26.3.6 RTC_SET_INTERRUPT_DISABLE - disable RTC interrupt

Call(s):

```
void ioctl(const int *pModuleBase, RTC_SET_INTERRUPT_DISABLE,
           NULL);
```

Arguments:

Table 5-934. RTC_SET_INTERRUPT_DISABLE ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_SET_INTERRUPT_DISABLE* ioctl command disables the RTC interrupt.

This command writes directly into the RTC Real-Time Interrupt Enable (RTIE) bit of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_SET_INTERRUPT_DISABLE* command is implemented as a macro.

Example 5-818. RTC_SET_INTERRUPT_DISABLE

```
ioctl(RTC, RTC_SET_INTERRUPT_DISABLE, NULL);
```

This code disables the RTC Real-Time interrupt.

5.26.3.7 RTC_TEST_INTERRUPT_FLAG - test RTC interrupt flag

Call(s):

```
UWord16 ioctl(const int *pModuleBase, RTC_TEST_INTERRUPT_FLAG,
          NULL);
```

Arguments:

Table 5-935. RTC_TEST_INTERRUPT_FLAG ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_TEST_INTERRUPT_FLAG* ioctl command tests the RTC interrupt flag.

This status bit indicates that the RTC counter register reached the value in the RTC modulo register. Writing a logic one clears the bit and the real-time interrupt request.

This command reads directly from the RTC Real-Time Interrupt Flag (RTIF) bit of the RTC Status and Control Register.

Returns: UWord16 value of RTC interrupt flag.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_TEST_INTERRUPT_FLAG* command is implemented as a macro.

Example 5-819. RTC_TEST_INTERRUPT_FLAG

```
if (ioctl(RTC, RTC_TEST_INTERRUPT_FLAG, NULL))
{
      ...
}
```

This code tests the RTC Real-Time interrupt flag.

5.26.3.8 RTC_CLEAR_INTERRUPT_FLAG - clear RTC interrupt flag

Call(s):

```
void ioctl(const int *pModuleBase, RTC_CLEAR_INTERRUPT_FLAG,
           NULL);
```

Arguments:

Table 5-936. RTC_CLEAR_INTERRUPT_FLAG ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_CLEAR_INTERRUPT_FLAG* ioctl command clears the RTC interrupt flag.

This command clears directly the RTC Real-Time Interrupt Flag (RTIF) bit of the RTC Status and Control Register.

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_CLEAR_INTERRUPT_FLAG* command is implemented as a macro.

Example 5-820. RTC_CLEAR_INTERRUPT_FLAG

```
if (ioctl(RTC, RTC_TEST_INTERRUPT_FLAG, NULL))
{
      ...
}
ioctl(RTC, RTC_CLEAR_INTERRUPT_FLAG, NULL);
```

This code shows how pending interrupts are read, processed and cleared.

5.26.3.9 RTC_WRITE_MODULO_REG - set RTC counter terminal value

Call(s):

```
void ioctl(const int *pModuleBase, RTC_WRITE_MODULO_REG,
           UWord16 param);
```

Arguments:

Table 5-937. RTC_WRITE_MODULO_REG ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.
param	in	UWord16 modulo value.

Description: The *RTC_WRITE_MODULO_REG* ioctl command writes parameter into the counter modulo register, which presents the terminal value of the RTC counter. When counter reaches the modulo value, it is wrapped to zero at next counter clock and the RTC interrupt event is raised (if enabled). Modulo value of zero causes the RTC counter to remain zero and to generate the RTC interrupts at each counter clock (clock divided by prescaler value).

This command writes directly into the Modulo-Value Register (MOD).

Returns: None.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_WRITE_MODULO_REG* command is implemented as a macro.

Example 5-821. RTC_WRITE_MODULO_REG

```
ioctl(RTC, RTC_SET_CLOCK_SOURCE_PRESCALER,
      RTC_CLOCK_INT_DIV_128);
ioctl(RTC, RTC_WRITE_MODULO_REG, 249);
```

This code configures RTC module to generate 1kHz periodic interrupts at 32MHz IP Bus Clock.

5.26.3.10 RTC_READ_MODULO_REG - read RTC counter terminal value

Call(s):

```
UWoprd16 ioctl(const int *pModuleBase, RTC_READ_MODULO_REG,NULL);
```

Arguments:

Table 5-938. RTC_READ_MODULO_REG ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_READ_MODULO_REG* ioctl command reads and returns the Modulo Value Register content.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_READ_MODULO_REG* command is implemented as a macro.

Example 5-822. RTC_READ_MODULO_REG

```
UWord16 mod;

mod = ioctl(RTC, RTC_READ_MODULO_REG, NULL);
```

This code reads the RTC modulo value.

5.26.3.11 RTC_READ_COUNTER_REG - read RTC counter value

Call(s):

```
UWord16 ioctl(const int *pModuleBase, RTC_READ_COUNTER_REG,NULL);
```

Arguments:

Table 5-939. RTC_READ_COUNTER_REG ioctl call arguments

*pModuleBase	in	The RTC module identifier. Use RTC.

Description: The *RTC_READ_COUNTER_REG* ioctl command reads and returns the RTC Counter register content.

Returns: The register value as UWord16.

Range Issues: None.

Special Issues: This command is applicable only on MC56F800x.

Design/Implementation: The *RTC_READ_COUNTER_REG* command is implemented as a macro.

Example 5-823. RTC_READ_COUNTER_REG

```
UWord16 count;

count = ioctl(RTC, RTC_READ_COUNTER_REG, NULL);
```

This code reads the immediate counter value of RTC.

5.26.4 RTC Driver Applications

There is onee sample applications demonstrating how RTC module can be used in a user application. The supported DEMO boards for RTC sample application are MC56F8006DEMO.

5.26.4.1 rtc_demo

{DSP56800E_Quick_Start Source}\..\sample_applications\MC56F8006DEMO***rtc_demo***

Sample application which demostrates how RTC module interrupt can be handled. The sample application is using two LED diodes. One Led diode indicates that RTC interrupt occured and second LED diode indicates that main loop is working correctly. This application is using FreeMASTER, where is monitored counter value. In FreeMASTER can setted up RTCMOD register, prescaler and clock source.

Chapter 6
FreeMASTER Driver

6.1 Introduction

The FreeMASTER driver handles a communication between embedded application and the PC running the FreeMASTER tool. A communication protocol and the FreeMASTER tool enables real-time access to target resources, especially C-language variables and memory in general.

FreeMASTER protocol features:

- Read/Write Access to any memory location on the target.
- Atomic bit manipulation on target memory (bit-wise write access).
- Oscilloscope access - optimized real time access to variables (up to 8 variables). Sample rate depends on communication speed.
- Recorder - access to fast transient recorder running on-board as a part of FreeMASTER driver. Sample rate is limited by microcontroller CPU speed only. The length of data recorded depends on amount of available memory, 64kB maximum.
- Application commands - high level message delivery from PC to the application.

FreeMASTER driver features:

- Full FreeMASTER protocol implementation.
- SCI or JTAG as a native communication interface.
- Ability to write-protect memory regions or individual variables.
- Ability to deny access to unsafe memory.
- Two ways of how to handle Application Commands
  - — Classic: the application polls the App.Command status to determine any command is pending.
  - — Callback: the application registers a callback function which is automatically invoked upon reception of given command.
  - — The two approaches may be mixed in the application. Callback commands do not appear in the polling mechanism.

In DSP56800E_Quick_Start r2.1, the FreeMASTER driver was completely re-written to enable better code portability and memory protection features. The former Quick Start versions included similar driver named “pc_master”, which is still available for backward compatibility.

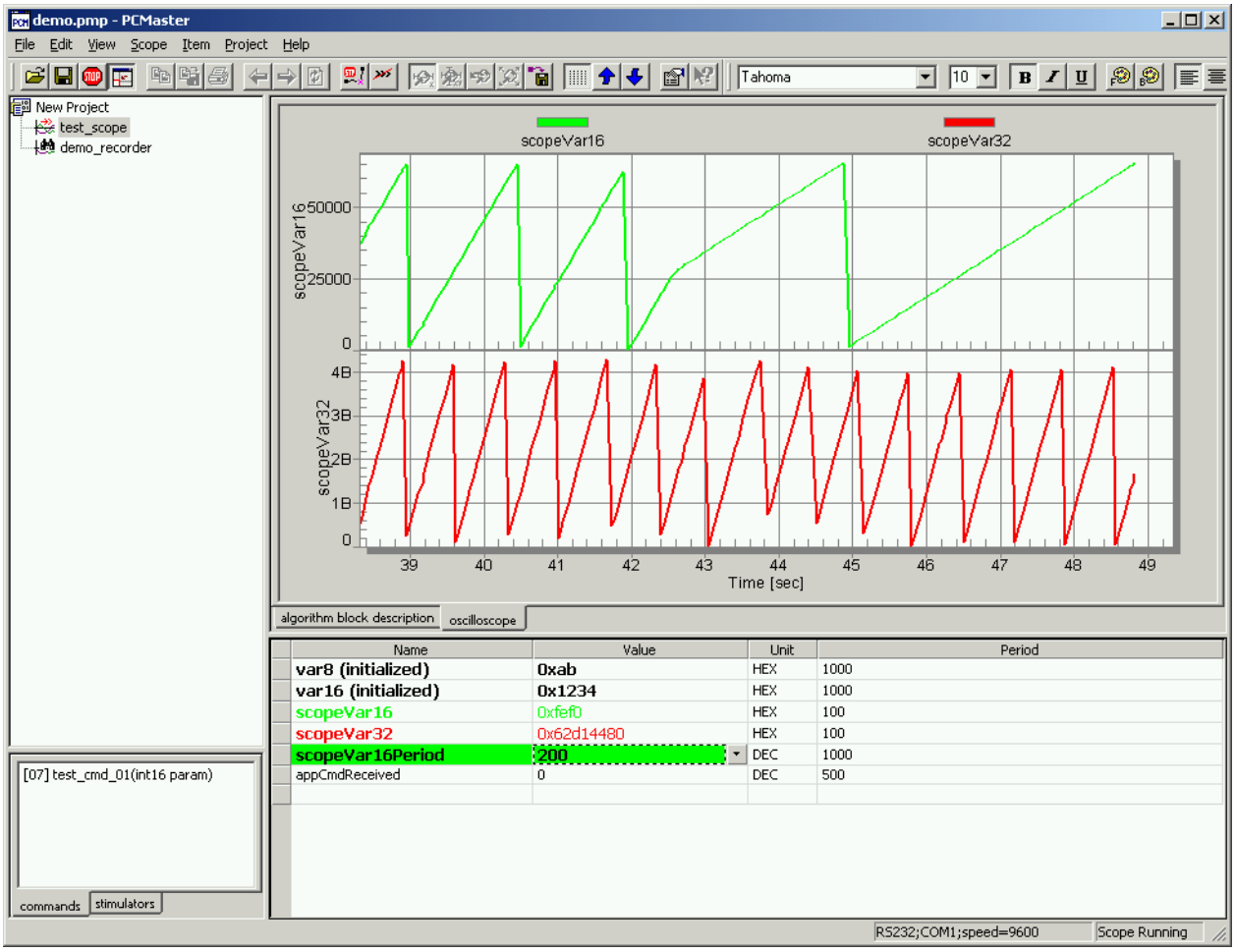


Figure 6-1. FreeMASTER Application Window

6.2 Driver Files

The driver code can be found in the `src/support/freemaster` subdirectory of the main install directory. The driver is highly configurable and not all of the .c files are needed to be compiled in all configurations. However, *all* the files may be added to the project as there are conditional compilation statements in the code, which assure only the required code gets actually compiled.

The driver is configured by macro constants defined in the *freemaster_cfg.h* file. This file is included by all FreeMASTER driver source files and should contain configuration constants which affect how the driver is compiled. With the DSP56800E_Quick_Start, the *freemaster_cfg.h* file is rather simple and is not to be modified. The actual configuration should be done in the Quick Start's *appconfig.h* file which is included by *freemaster_cfg.h*. This scenario enables the FreeMASTER driver to be configured graphically, using the Graphical Configuration Tool.

The FreeMASTER driver may also be used without the DSP56800E_Quick_Start and thus also without the *appconfig.h* file. In this case, all the configuration constants should be placed directly to the freemaster_cfg.h.

The following files are located in the `src/support/freemaster` directory:

- **freemaster.h** - The main driver header file. This is the only file which needs to be included by the application code.
- **freemaster_cfg.h** - The minimal driver configuration file which further includes the **appconfig.h** file and enables the driver to be configured in Graphical Configuration Tool.
- **freemaster_cfg.h.template** - An example of the full driver configuration file. Such a file needs to be used (and renamed to *freemaster_cfg.h*) in case the FreeMASTER driver is extracted and used outside the DSP56F800E_Quick_Start environment.
- **freemaster_private.h** - Internal header file used by all driver source code. The compile-time verification of the configuration macros is done in this file. Also the default values of undefined configuration values are defined here.
- **freemaster_56F8xxx.c** - 56F8xxx hardware-specific functions are implemented in this file.
- **freemaster_56F8xxx.h** - 56F8xxx hardware-specific macros and inline functions are in this file.
- **freemaster_protocol.c** - Implements the FreeMASTER protocol decoder, which is independent on the communication interface used.
- **freemaster_protocol.h** - Internal header file which contains the FreeMASTER protocol constants.
- **freemaster_serial.c** - Physical SCI and JTAG interface is handled in this file. The protocol decoder is invoked from this file when a valid FreeMASTER message is received. This file also handles the response transmission back to the PC.
- **freemaster_appcmd.c** - Implementation of the FreeMASTER Application Commands.
- **freemaster_rec.c** - FreeMASTER Recorder implementation.
- **freemaster_scope.c** - FreeMASTER Oscilloscope implementation.
- **freemaster_tsa.c** - Target-side Addressing (TSA) implementation.
- **freemaster_tsa.h** - TSA header file which is indirectly included in the user application code too. This file defines the macros which are used to build the TSA tables in the user code.

6.3 Interrupt Handling

The FreeMASTER driver must be configured for one of the three modes of operation, differing in the way how the peripheral interrupts are used by the driver.

Table 6-1. FreeMASTER Driver Interrupt Mode

Mode	Description
Long Interrupt Mode	Both the communication interface and the protocol decoder are processed in the SCI or JTAG interrupt. The time spent in the interrupt routine depends on the what protocol command is being handled. Such a non-deterministic behavior may require the application interrupt levels to be balanced with care.
Short Interrupt Mode	The communication is handled in the interrupt while the protocol is decoded and handled only when the “poll” function is called. The time spent in the interrupt service routine is rather short as the characters are just fetched to or from the buffer. One additional communication buffer (receiver queue) is required to store received characters before they are handled by the protocol state machine.

Table 6-1. FreeMASTER Driver Interrupt Mode (Continued)

Mode	Description
Poll-driven Mode	No-interrupt operation. Both the communication interface and the protocol are handled only when the “poll” function is called. Typically, this function is called from the application main loop.

6.4 New Features

In addition to the JTAG communication support, there are two other important new features in the FreeMASTER driver.

6.4.1 Target-side Addressing

One of the new features in the FreeMASTER driver if comparing it with the old “pc_master” driver is the Target-side Addressing capability (TSA). With this feature, the user is able to describe the variables and structure data types directly in the application source code and make this information available for the FreeMASTER tool. The tool may then use this information instead of reading it from the application’s ELF/Dwarf executable file.

Once the variables are described in TSA tables in the application source code (see Section 6.6.1, “Driver API.” below), the FreeMASTER driver can also use this information to protect other memory from being accessed by the PC.

6.4.2 Application Command Callbacks

Another new feature of the FreeMASTER driver is a capability of invoking the user-defined callback functions when the Application Command is received. The callback function obtains the Application Command number and data as a parameters. Callback’s return value is used as a Application Command result code.

In the old driver, the Application Command status needed to be periodically tested, for example in the application main loop. This approach is still possible with the FreeMASTER driver.

6.5 Driver Configuration

This section describes the FreeMASTER driver configuration constants which can be placed in the *appconfig.h* file (or *freemaster_cfg.h* file).

Table 6-2. FreeMASTER Communication Configuration Items for appconfig.h

SYMBOL	TYPE	DESCRIPTION
FMSTR_LONG_INTR FMSTR_SHORT_INTR FMSTR_POLL_DRIVEN	numeric (zero / non-zero)	Exactly one of the three constants must be set to a non-zero value. Such a macro then affects how the FreeMASTER driver uses interrupts. See table Table 6-1 above.
FMSTR_SCI_BASE	address (numeric)	In case the SCI is to be used as a communication interface, this constant must be defined as a base address of SCI peripheral register space. With DSP56800E_Quick_Start, you can find these constants in the processor-specific “arch.h” file (SCI0_BASE, SCI1_BASE or SCI_BASE).
FMSTR_USE_JTAG	numeric (zero / non-zero)	When defined as a non-zero value, the JTAG will be used for communication instead of the SCI. **Default: 0 (false)**
FMSTR_USE_JTAG_TXFIX	numeric (zero / non-zero)	When non-zero, the driver implements software workaround of the JTAG TDF status bit problem. See more information in the JTAG communication plug-in for the FreeMASTER tool. **Default: 0 (false)**
FMSTR_JTAG_BASE	address (numeric)	In case the JTAG is to be used as a communication interface, this constant may be defined as a base address of JTAG register space. As the JTAG registers are located at the same address on all 56F8xxx processors, this value needs not to be defined and its **default** value is **0xFFFF00**.
FMSTR_COMM_BUFFER_SIZE	integer	Defines the size of the FreeMASTER communication buffer. This buffer is used to store the frames being received or frames to be sent over the SCI or JTAG. If this constant is undefined or is defined as zero, a sufficient memory buffer is allocated by the driver. **Default: 0 (automatic)**
FMSTR_COMM_RQUEUE_SIZE	integer	The size of additional receive queue for short interrupt processing. Needed only in FMSTR_SHORT_INTR mode. **Default: 32**
Application Commands		
FMSTR_USE_APPCMD	numeric (zero / non-zero)	When defined non-zero, this constant enables a support for FreeMASTER Application Commands. **Default: 0 (false)**

Table 6-2. FreeMASTER Communication Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
FMSTR_APPCMD_BUFF_SIZE	integer	A size of buffer which will receive the Application Command parameters. The parameters are stored in the buffer until the application code processes and acknowledges the application command. **Default: 16**
FMSTR_MAX_APPCMD_CALLS	integer	A number of different Application Commands for which the callback functions are to be registered. If this constant is not defined or is zero, the support for Application Command callbacks is not compiled in. **Default: 0**
Oscilloscope		
FMSTR_USE_SCOPE	numeric (zero / non-zero)	When defined non-zero, this constant enables a support for the FreeMASTER Oscilloscope. Oscilloscope enables a faster (simultaneous) access to up to 8 variables and is used when the variables are to be displayed in the real-time graph. If this feature is not enabled, the FreeMASTER uses standard memory access commands to access variables in graph sequentially. **Default: 0 (false)**
FMSTR_MAX_SCOPE_VARS	integer	This constant defines how many variables will it be possible to display in the FreeMASTER Oscilloscope. The **default** value is **8**. It makes sense to set this value in the range of 2...8. Values lower than 8 may save some data RAM allocated by the driver (6bytes per variable). The current version of the FreeMASTER tool does not support more than 8 variables to be displayed in graph.
Recorder		
FMSTR_USE_RECORDER	numeric (zero / non-zero)	When defined non-zero, this constant enables a support for the FreeMASTER Recorder. Recorder is an advanced feature which enables up to 8 variables to be internally sampled to the on-board memory. The sampling can be stopped manually or automatically based on a threshold crossing condition. Sampled data buffer is downloaded to the FreeMASTER as a block and data values are displayed in the graph. Disabling the recorder feature may save significant amount of code and data memory. **Default: 0 (false)**

Table 6-2. FreeMASTER Communication Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
FMSTR_MAX_REC_VARS	integer	This constant defines how many variables will it be possible to display in the FreeMASTER Recorder. The **default** value is **8**. It makes sense to set this value in the range of 2...8. Values lower than 8 may save some data RAM allocated by the driver (6bytes per variable). The current version of the FreeMASTER tool does not support more than 8 variables to be displayed in graph.
FMSTR_REC_OWNBUFF	numeric (zero / non-zero)	When defined non-zero, the user may (must) supply his own memory to be used as a recorder buffer. The recorder buffer is the memory dedicated to the recorder feature and is used to hold sampled variable values. As the buffer may be up to 64kB long, it may sometimes be desirable the user allocates the memory himself and do not let the driver to allocate it statically. When undefined or set to zero, the FreeMASTER driver statically allocates the recorder buffer (FMSTR_REC_BUFF_SIZE bytes). **Default: 0 (false)**
FMSTR_REC_BUFF_SIZE	integer UWord16	This constant is used when FMSTR_REC_OWNBUFF is undefined or set to zero. It defines the size of the recorder buffer which is to be allocated by the FreeMASTER driver. **Default: 256**
FMSTR_REC_FARBUFF	numeric (zero / non-zero)	This option is specific to the 56F83xx family. When defined non-zero, the recorder buffer allocated by the FreeMASTER driver is put in the ".fardata" memory segment, which is typically put to the external memory after the address 0x10000 by the linker. **Default: 0 (false)**

Table 6-2. FreeMASTER Communication Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
FMSTR_REC_TIMEBASE	UWord16 constant	This constant is used to describe the recorder sampling rate as it is implemented in the application. The time base value specified by this constant is used as the base X-axis unit in the recorder graph. You can use one of the following macros to build the value (providing a 14bit 'x" value): - FMSTR_REC_BASE_NANOSEC(x) - FMSTR_REC_BASE_MICROSEC(x) - FMSTR_REC_BASE_MILLISEC(x) - FMSTR_REC_BASE_SECONDS(x) The (**default**) value of **zero** can be used as "unknown", which forces the FreeMASTER recorder to display the X-axis values as indexes instead of time.
Target-side Addressing		
FMSTR_USE_TSA	numeric (zero / non-zero)	When defined non-zero, this constant enables a support for so-called Target-side Addressing (TSA). This is a new feature in the FreeMASTER protocol, which enables the FreeMASTER tool to obtain variable information directly from the embedded application. With this feature, it is no more necessary for the FreeMASTER tool to load and parse the application ELF file. It also enables the TSA-safety feature. **Default: 0 (false)**
FMSTR_USE_TSA_SAFETY	numeric (zero / non-zero)	When defined non-zero and TSA is enabled, this constant activates the FreeMASTER memory protection. In this mode, the driver actively denies any access to memory areas which are not described by any TSA table in the application. Also with this feature, the variables described by a TSA entries can be declared as Read/Write or Read-Only for the FreeMASTER tool. **Default: 0 (false)**
FMSTR_USE_TSA_INROM	numeric (zero / non-zero)	With this constant defined as a non-zero value, all TSA tables are declared as "const", i.e they are put to ".const.data" segment. This segment is typically put to Flash memory in DSP56F800E_Quick_Start applications. **Default: 0 (false)**

Table 6-2. FreeMASTER Communication Configuration Items for appconfig.h (Continued)

SYMBOL	TYPE	DESCRIPTION
Memory Access		
FMSTR_USE_READMEM	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Read-Memory feature is implemented. It is recommended to enable this feature. **Default: 1 (true)**
FMSTR_USE_WRITEMEM	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Write-Memory feature is implemented. It is recommended to enable this feature. **Default: 1 (true)**
FMSTR_USE_WRITEMEMMASK	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Masked (bit-wise) Memory Write feature is implemented. It is recommended to this feature. **Default: 1 (true)**
Variable Access *(subset of Memory Access features, implemented for backward compatibility only)*		
FMSTR_USE_READVAR	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Read-Variable feature is implemented. This command enables a read access to 1, 2 or 4 byte variables. This functionality is a subset of the Read-Memory feature. Comparing it with the Read-Memory feature, there is one byte saved on the communication line. **Default: 0 (false)**
FMSTR_USE_WRITEVAR	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Write-Variable feature is implemented. This command enables a write access to 1, 2 or 4 byte variables. This functionality is a subset of the Write-Memory feature. Comparing it with the Write-Memory feature, there is one byte saved on the communication line. **Default: 0 (false)**
FMSTR_USE_WRITEVARMASK	numeric (zero / non-zero)	When this constant is defined as a non-zero value, the support for Masked Write-Variable feature is implemented. This command enables a write access to 1 and 2 byte variables only. This functionality is a subset of the Masked Write-Memory feature. Comparing it with the Masked Write-Memory feature, there is one byte saved on the communication line. **Default: 0 (false)**

6.6 Driver Usage

The FreeMASTER driver was designed to be easily used in any DSP56F800E_Quick_Start application and also in custom user code. With DSP56F800E_Quick_Start, the following steps are necessary to enable a basic FreeMASTER connectivity in the application:

- The FreeMASTER driver needs to be configured in the **Graphical Configuration Tool** or in the *appconfig.h* file directly.
- The **freemaster.h** file needs to be included in any application source file which makes FreeMASTER API calls.
- The **FMSTR_Init()** function has to be called before any other FreeMASTER driver API calls.
- For FMSTR_LONG_INTR or FMSTR_SHORT_INTR modes, the **SCI or JTAG interrupts** need to be directed to **FMSTR_Isr()** function. This is done automatically when the FreeMASTER driver is configured using the Graphical Configuration Tool.

  The **interrupt priority levels** need to be set (in GCT) and the interrupts have to be enabled in the processor core (**archEnableInt()**).
- For FMSTR_SHORT_INTR or FMSTR_POLL_DRIVEN modes, the **FMSTR_Poll()** API function needs to be called periodically in the application. For example in the application main loop.

See Driver API description in the next section for more detail on how to use advanced FreeMASTER features like Application Commands, Recorder or TSA.

6.6.1 Driver API

This section describes the FreeMASTER driver API. It can be expected the same API will be used with the new FreeMASTER driver on other Freescale platforms as well.

The following header files are needed in order to use the FreeMASTER driver:

Required Header File(s):

```
#include "qs.h"
#include "freemaster.h"
```

The next sections describe each driver API function in detail. Function arguments for each routine are described as *in* or *out*

1. *in* argument means that the parameter value is an input only to the function.
2. *out* argument means that the parameter value is an output only from the function.

6.6.1.1 FMSTR_Init() - Initializes the FreeMASTER Communication

Call:

```
void FMSTR_Init(void);
```

Arguments: None.

Description: This function initializes internal variables of the FreeMASTER driver and enables the communication interface (SCI or JTAG). This function does not change the configuration of communication module, the module must be initialized before the *FMSTR_Init* function is called.

For the SCI communication, the SCI module has to be configured either dynamically in run-time or using the GCT and the *SCI_INIT* ioctl call. See Section 5.15.3.1 for more details. It is not necessary to set the transmitter and receiver enable bits in the SCI control register (SCICR).

In case the interrupt-driven SCI communication is to be used either in *FMSTR_SHORT_INTR* or *FMSTR_LONG_INTR* mode, the SCI interrupt vectors need to be routed to *FMSTR_Isr* function and interrupt priority levels should be set to Level 0, 1, or 2. It is not necessary to set any of the interrupt enable bits in the SCI control register (SCICR). The SCI TX Idle interrupt is not used at all.

For the JTAG communication (as there is no settings for the JTAG module) the only what needs to be done is routing the JTAG interrupt vectors to *FMSTR_Isr* function and setting proper interrupt priority levels. For JTAG polling mode (*FMSTR_POLL_DRIVEN*), no setup operations are required at all.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-1. FMSTR_Init

```
...
ioctl(SCI_0, SCI_INIT, NULL);
...
...
FMSTR_Init();
...
```

This code initializes the SCI module and then it initializes the FreeMASTER driver

6.6.1.2 FMSTR_Poll() - The FreeMASTER polling method

Call:

```
void FMSTR_Poll(void);
```

Arguments: None.

Description: Except when the driver is configured for the *FMSTR_LONG_INTR* mode, the *FMSTR_Poll* function must be called periodically in the application. Typically, such a call is placed to the application main loop, while the application critical tasks are handled in the interrupts.

In the *FMSTR_SHORT_INTR* interrupt mode, the *FMSTR_Poll* function fetches all bytes which were received by the SCI or JTAG interrupt service routine and were put in the temporary receive queue. The fetched bytes are then all fed to the protocol state machine. If the state machine determines the protocol message is received without error, the protocol message decoder is invoked. If, in-turn, the message is evaluated as a valid and supported message, the protocol message handler function is invoked to process the message.

To prevent receive data overflow in *FMSTR_SHORT_INTR* mode, the *FMSTR_Poll* function must be called at least once per: *FMSTR_COMM_RQUEUE_SIZE* * *Tchar* time, where *Tchar* is the time it takes to receive one byte.

In the *FMSTR_POLL_DRIVEN* mode, there is no receive queue and no interrupt service routine which would handle character reception. In this mode, the *FMSTR_Poll* function accesses the SCI or JTAG peripheral status registers to detect any character was received. To prevent receive data overflow, the *FMSTR_Poll* function must be called at least once per *Tchar* time.

In the *FMSTR_LONG_INTR* mode, the complete process described above is performed during the SCI or JTAG interrupt. There is no need to call the *FMSTR_Poll* function at all and it is actually compiled to an empty function. You can still have the function being called in the application main loop to make the code independent on interrupt mode selected.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-2. FMSTR_Poll

```
...
for(;;)
{
      ...
      FMSTR_Poll();
}
```

This code demonstrates the usage of the FMSTR_Poll API function

6.6.1.3 FMSTR_Isr() - The FreeMASTER interrupt dispatcher

Call:

```
#pragma interrupt
void FMSTR_Isr(void);
```

Arguments: None.

Description: Except when the driver is configured for the *FMSTR_POLL_DRIVEN* mode, the *FMSTR_Isr* function must be used as the SCI or JTAG interrupt service routine in the application.

This function handles the SCI or JTAG communication for the *FMSTR_SHORT_INTR* and *FMSTR_LONG_INTR* modes.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-3. FMSTR_Isr

```
#define INT_VECTOR_ADDR_68              FMSTR_Isr
#define INT_PRIORITY_LEVEL_68           INTC_LEVEL1
#define INT_VECTOR_ADDR_71              FMSTR_Isr
#define INT_PRIORITY_LEVEL_71           INTC_LEVEL1
#define INT_VECTOR_ADDR_72              FMSTR_Isr
#define INT_PRIORITY_LEVEL_72           INTC_LEVEL1
```

This (*appconfig.h*) code shows how the FMSTR_Isr is set as SCI_0 TX Empty, RX Full and RX Error interrupt service routine in the 56F8346 project.

6.6.1.4 FMSTR_Recorder() - The FreeMASTER recorder engine

Call:

```
#pragma interrupt called
void FMSTR_Recorder(void);
```

Arguments: None.

Description: This function takes one sample of variable values being recorded using the FreeMASTER recorder. In case the recorder is not active at the moment when *FMSTR_Recorder* is called, the function returns immediately. When the recorder is active, the values of variables being recorded are copied to the recorder buffer and the trigger condition is evaluated.

In case the trigger condition is satisfied, the recorder enters the post-trigger mode, in which it counts the follow-up samples (*FMSTR_Recorder* function calls) and de-activates the recorder when required post-trigger samples are sampled.

Typically, you call the *FMSTR_Recorder* function in the Timer or PWM interrupt service routine. For simple test purposes, this function can also be called in the application main loop.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: This function saves and restores all registers used, so it is possible to invoke this function from an interrupt service routine safely.

Example 6-4. FMSTR_Recorder

```
#pragma interrupt
void timer_isr()
{
      ...
      FMSTR_Recorder();
      ...
}
```

This code shows how the FMSTR_Recorder function is invoked as a part of Timer interrupt service routine.

6.6.1.5 FMSTR_TriggerRec() - Force recorder trigger

Call:

```
void FMSTR_TriggerRec(void);
```

Arguments: None.

Description: This function forces the recorder trigger condition to happen, which causes the recorder to be automatically de-activated after a post-trigger samples are sampled. This function can be used in the application when it needs to have the trigger occurrence under its control.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-5. FMSTR_TriggerRec

```
FMSTR_TriggerRec();
```

This code shows how the FMSTR_TriggerRec function is invoked

6.6.1.6 FMSTR_SetUpRecBuff() - Set the recorder buffer

Call:

```
void FMSTR_SetUpRecBuff(
      FMSTR_ADDR nBuffAddr,
      FMSTR_SIZE nBuffSize);
```

Arguments:

- *nBuffAddr* - a pointer to the memory which is to be used as a recorder buffer
- *nBuffSize* - a size of the memory buffer

Description: This function can only be used when the *FMSTR_REC_OWNBUFF* configuration constant is set to a non-zero value. The user calls this function to "give" the data buffer he allocated to the FreeMASTER driver, which will use it as a recorder buffer.

Up to 64kB buffer may be used as a recorder buffer.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-6. FMSTR_SetUpRecBuff

```
UWord16 my_rec_buff[0x1000];

...
...

FMSTR_SetUpRecBuff(my_rec_buff, 0x1000);
```

This code shows how the user-allocated buffer can be set as the recorder buffer.

6.6.1.7 FMSTR_GetAppCmd() - Get pending Application Command

Call:

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

Arguments: None.

Description: This function can be used to detect if any Application Command is waiting to be processed by the application. In case there is no command pending, this function returns *FMSTR_APPCMDRESULT_NOCMD* constant. Otherwise, this function returns a code of the Application Command which needs to be processed. Use the *FMSTR_AppCmdAck* API call to acknowledge the Application Command after it is processed and to return the appropriate result code to the FreeMASTER tool.

The *FMSTR_GetAppCmd* function does not report commands for which a callback handler function exists. In case the *FMSTR_GetAppCmd* function is called when a callback-registered Command is pending (and before it is actually processed by the callback function), this function returns *FMSTR_APPCMDRESULT_NOCMD*.

Returns: The code of an Application Command which needs to be processed.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-7. FMSTR_GetAppCmd

```
...
// check if a new command has been received
nAppCmdCode = FMSTR_GetAppCmd();

// when a new command arrives, the nAppCmdCode contains command
// code. In other case, the "NOCMD" special value is returned
if (nAppCmdCode != FMSTR_APPCMDRESULT_NOCMD)
{
      // each command may have different processing and different
      // result code. The command processing is finished by
      // calling FMSTR_AppCmdAck() with the result code value
      switch(nAppCmdCode)
      {
      case 1:
            ...
            ...
            FMSTR_AppCmdAck(0x13);
            break;
      case 2:
            ...
```

This code shows how to process the Application Commands, for example in the main loop.

6.6.1.8 FMSTR_GetAppCmdData() - Get Application Command data

Call:

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* pDataLen);
```

Arguments:

- *pDataLen* (out) - pointer to a variable which receives the length of the data available in the buffer. May be NULL when this information is not needed.

Description: This function can be used to retrieve the Application Command data, once the application determines the Application Command is pending (see *FMTR_GetAppCmd* function above).

There is just a single buffer to hold the Application Command data (the buffer length is *FMSTR_APPCMD_BUFF_SIZE* bytes). In case the data are to be used in the application after the command is processed by the *FMSTR_AppCmdAck* call, the user needs to copy the data out to a private buffer.

Returns: Pointer to Application Command data.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-8. FMSTR_GetAppCmdData

```
...
      switch(nAppCmdCode)
      {
      case 1:
            ...
            pData = FMSTR_GetAppCmdData(NULL);
            ...
            FMSTR_AppCmdAck(0x13);
            break;
      case 2:
            ...
```

This code shows how to get the Application Command data when processing the Command.

6.6.1.9 FMSTR_AppCmdAck() - Finish processing of the Application Command

Call:

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT nResultCode);
```

Arguments:

- *nResultCode* (in) - the result code which is to be returned to the FreeMASTER tool

Description: This function is used when Application Command processing is finished in the application. The *nResultCode* passed to this function is returned back to the FreeMASTER tool and the driver is re-initialized to expect other Application Commands.

After this function is called, and before the next Application Command arrives, the return value of the *FMSTR_GetAppCmd* function is *FMSTR_APPCMDRESULT_NOCMD*.

Returns: None.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-9. FMSTR_AppCmdAck

```
...
switch(nAppCmdCode)
{
case 1:
      ...
      pData = FMSTR_GetAppCmdData(NULL);
      ...
      FMSTR_AppCmdAck(0x13);
      break;
case 2:
      ...
```

This code shows how to finish an Application Command processing.

6.6.1.10 FMSTR_AppCmdSetResponseData() - Set Application Command response data

Call:

```
void FMSTR_AppCmdSetResponseData(
      FMSTR_ADDR nResultDataAddr,
      FMSTR_SIZE nResultDataLen);
```

Arguments:

- *nResultDataAddr* (in) - pointer to data buffer which is to be copied to the Application Command data buffer
- *nResultDataLen* (in) - the length of a data to be copied. Must not exceed the *FMSTR_APPCMD_BUFF_SIZE* value.

Description: This function can be used *before* the Application Command processing is finished, when there are any data to be returned back to the PC. This function exists as such a feature is enabled by the FreeMASTER protocol. However, the current version of the FreeMASTER tool does not support the Application Command response data.

Returns: None.

Range Issues: None.

Special Issues: The response data buffer is copied to the Application Command data buffer, from where it is accessed in case the host requires it. Do not use the *FMSTR_GetAppCmdData* and the data buffer after the *FMSTR_AppCmdSetResponseData* is called.

Design/Implementation: None.

Example 6-10. FMSTR_GetAppCmdData

```
...
      switch(nAppCmdCode)
      {
      case 1:
            ...
            FMSTR_AppCmdSetResponseData("Hello", 5);
            ...
            FMSTR_AppCmdAck(0x13);
            break;
      case 2:
            ...
```

This code shows how to return a response data before the Application Command is finished.

6.6.1.11 FMSTR_RegisterAppCmdCall() - Register Application Command callback function

Call:

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(
      FMSTR_APPCMD_CODE nAppCmdCode,
      FMSTR_PAPPCMDFUNC pCallbackFunc);
```

Arguments:

- *nAppCmdCode* (in) - an Application Command code for which the callback is to be registered
- *pCallbackFunc* (in) - pointer to a callback function which is to be registered. Use NULL to un-register a callback registered previously with this Application Command.

Description: This function can be used to register a given function as a callback handler for an Application Command. The Application Command is identified using the single-byte code. The callback function is invoked automatically by the FreeMASTER driver, when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function should be:

```
FMSTR_APPCMD_RESULT HandlerFunction(
      FMSTR_APPCMD_CODE nAppcmd,
      FMSTR_APPCMD_PDATA pData,
      FMSTR_SIZE nDataLen)
```

Where

- *nAppcmd* - is the Application Command code
- *pData* - points to Application Command data received (if any)
- *nDataLen* - is the information about Application Command data length

The return value of the callback function is used as the Application Command result code and is returned to FreeMASTER tool.

Returns: Non-zero value when the callback function was successfully registered.

Range Issues: None.

Special Issues: None.

Design/Implementation: None.

Example 6-11. FMSTR_RegisterAppCmdCall

```
...
FMSTR_RegisterAppCmdCall(10, HandlerFunction);
```

This code shows how to register a callback function for Application Command with code 10.

6.6.1.12 Target-side Addressing API

When the Target-side Addressing (TSA) is enabled in the FreeMASTER driver configuration file (by setting the *FMSTR_USE_APPCMD* constant non-zero), the user should define so-called TSA tables in his application. This section describes macros which need to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There should be always one TSA Table List defined which informs the FreeMASTER driver about TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information should automatically appear in the FreeMASTER symbols list. The FreeMASTER user is then able to create FreeMASTER variables based on these symbols. The TSA is supported in FreeMASTER version 1.2.37 and higher.

See the TSA example in the **freemaster_demo2 sample application** in DSP56800E_Quick_Start installation. The application is also printed below on page 6-27.

6.6.1.12.1 TSA Table Definition

The TSA table describes the static or global variables, together with their address, size, type and access-protection information. In case the TSA-described variables are of a structure type, the TSA table may also describe this type and enable FreeMASTER user to access individual structure members of the variable.

The TSA table definition begins with FMSTR_TSA_TABLE_BEGIN macro:

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

Where the *table_id* is any valid C-language symbol identifying the table. There can be any number of TSA tables in the application, provided the table identifiers remain unique.

After this macro, the TSA table entries are placed using one of the macros below:

```
FMSTR_TSA_RW_VAR(name, type)      // read/write variable entry
FMSTR_TSA_RO_VAR(name, type)      // read-only variable entry
FMSTR_TSA_STRUCT(struct_name)     // structure type entry
FMSTR_TSA_MEMBER(struct_name, member_name, type)
                                  // structure member entry
```

The table is finished using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

The TSA entry definition macros accept the following parameter:

- **`name`** - Variable name. The variable must be defined before the TSA table entry references it.
- **`type`** - Variable or member type. Only one of the pre-defined type constants may be used, as described below
- **`struct_name`** - Structure type name. The type must be defined (typedef) before the TSA table entry references it.
- **`member_name`** - Structure member name, without the dot at the beginning. The parent structure name is specified as a separate parameter in the FMSTR_TSA_MEMBER macro.

Note: Structure member entries (FMSTR_TSA_MEMBER) must immediately follow the parent structure entry (FMSTR_TSA_STRUCT).

Note: In order to write-protect variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), the FMSTR_USE_TSA_SAFETY needs to be defined non-zero in the configuration file.

Note: Despite of its name, the FMSTR_TSA_STRUCT macro may also be used to describe union data types.

Table 6-3. TSA Type Constants

CONSTANT	Description
FMSTR_TSA_UINT8 FMSTR_TSA_UINT16 FMSTR_TSA_UINT32 (FMSTR_TSA_UINT64)	1, 2, 4-byte unsigned integer type. Use it for both the standard C-language types like unsigned char, unsigned short or unsigned long and the user-defined types like UWord8, UWord16 or UWord32. 8-byte integer values are not used on 56F8xxx platform.
FMSTR_TSA_SINT8 FMSTR_TSA_SINT16 FMSTR_TSA_SINT32 (FMSTR_TSA_SINT64)	1, 2, 4-byte signed integer type. Use it for both the standard C-language types like char, short or long and the user-defined types like Word8, Word16 or Word32. 8-byte integer values are not used on 56F8xxx platform.
FMSTR_TSA_FRAC16 FMSTR_TSA_FRAC32	Fractional data types. Although these types are treated as integer types in C-language, it may be beneficial to describe them using these macro constants, so the FreeMASTER treats them properly.
FMSTR_TSA_UFRAC16 FMSTR_TSA_UFRAC32	Unsigned fractional data types. Although these types are treated as integer types in C-language, it may be beneficial to describe them using these macro constants, so the FreeMASTER treats them properly.
FMSTR_TSA_FLOAT	4-byte standard IEEE floating point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating point type
FMSTR_TSA_USERTYPE(**name**)	Structure or union type. You must specify the type name as an argument.

6.6.1.12.2 TSA Table List

There **must** be exactly one TSA Table List in the application. The list contains one entry for each TSA table which is defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro:

```
FMSTR_TSA_TABLE_LIST_BEGIN()
```

and continues with TSA table entries for each table:

```
FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is finished with the FMSTR_TSA_TABLE_LIST_END macro

```
FMSTR_TSA_TABLE_LIST_END()
```

6.6.2 Code Listing: freemaster_demo

The code listed below may be found in the FreeMASTER sample application main file for any 56F8xxx device.

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Simple sample application demonstrating the functionality of
*              FreeMASTER software driver. Please see also the freemaster_demo2
*              application which demonstrates use of advanced FreeMASTER
*              features (Target-side Addressing, Memory Access security etc.)
*
*              The FreeMASTER driver and the SCI module are configured by GCT
*
* TARGET: MC56F8xxx device
*
*****************************************************************************/

#include "qs.h"

#include "intc.h"
#include "gpio.h"
#include "occs.h"
#include "sci.h"
#include "sys.h"
#include "freemaster.h"

/* Test variables, will be displayed in the FreeMASTER application */
volatile UWord8  var8;
volatile UWord16 var16;
volatile UWord16 var16inc = 1;
volatile UWord32 var32;
volatile UWord32 var32inc = 100;

/* local prototypes */
void init_pins(void);
void init_uart(void);

////////////////////////////////////////////////////////////////////////

int main(void)
{
      UWord16 nAppCmdCode;

      // initialize SYS and GPIO modules
      init_pins();

      // initialize UART
      init_uart();

      // FreeMASTER initialization
      FMSTR_Init();

      // initialize & enable interrupts
      ioctl(INTC, INTC_INIT, NULL);
```

```
        archEnableInt();

        // other initializations
        var8 = 10;

        // main application loop
        while (1)
        {
                // scope variables
                var16 += var16inc;
                var32 += var32inc;

                // This call should be placed in the timer interrupt or anywhere where
                // the recorder sampling should occur.
                FMSTR_Recorder();

                // The FreeMASTER polling call must be called periodically in the main
                // application loop to handle the communication interface and protocol.
                // Only in LONG_INTR FreeMASTER interrupt mode, all the processing is
done
                // during the communication interrupt so the FMSTR_Poll() is not needed
                // (the function is compiled empty in this case).
                FMSTR_Poll();
        }
}

/*
 *  Initialize SYS and all GPIO modules.
 */

void init_pins(void)
{
        ioctl(SYS, SYS_INIT, NULL);

        // This code is targeted to all 56F8xxx-based evaluation boards.
        // On some processors, some GPIO instances are not implemented
#ifdef GPIO_A
        ioctl(GPIO_A, GPIO_INIT, NULL);
#endif
#ifdef GPIO_B
        ioctl(GPIO_B, GPIO_INIT, NULL);
#endif
#ifdef GPIO_C
        ioctl(GPIO_C, GPIO_INIT, NULL);
#endif
#ifdef GPIO_D
        ioctl(GPIO_D, GPIO_INIT, NULL);
#endif
#ifdef GPIO_E
        ioctl(GPIO_E, GPIO_INIT, NULL);
#endif
#ifdef GPIO_F
        ioctl(GPIO_F, GPIO_INIT, NULL);
#endif
}

/*
 *  Initialize the UART ports
 */

void init_uart(void)
{
        // initialize UART
        ioctl(SCI_0, SCI_INIT, NULL);

        // Initialize SCI_1 as well if such a module exists -
```

```
      // this may be useful on boards with SCI_1 RS232 port
#ifdef SCI_1
      ioctl(SCI_1, SCI_INIT, NULL);
#endif
}
```

The FreeMASTER driver configuration extracted from the appconfig.h file:

```
/*.
    FMSTR Configuration
-------------------------------------------
.*/
#define FMSTR_COMM_INTERFACE             1
#define FMSTR_LONG_INTR                  0
#define FMSTR_SHORT_INTR                 1
#define FMSTR_POLL_DRIVEN                0
#define FMSTR_USE_APPCMD                 1
#define FMSTR_USE_SCOPE                  1
#define FMSTR_USE_RECORDER               1
```

6.6.3 Code Listing: freemaster_demo2

The second FreeMASTER sample application demonstrates a usage of the TSA feature and both the callback and polled handling of the FreeMASTER Application Command.

```
/******************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2004 Freescale Semiconductor, Inc.
* (c) Copyright 2001-2004 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
******************************************************************************
*
* FILE NAME: main.c
*
* DESCRIPTION: Advanced sample application demonstrating the functionality of
*              FreeMASTER software driver. See also the freemaster_demo
*              application, which demonstrates the key FreeMASTER features
*              only (without TSA, security and app.cmd callbacks)
*
*              The FreeMASTER driver and the SCI module are configured by GCT
*
* TARGET: MC56F8xxx device
*
******************************************************************************/

#include "qs.h"

#include "intc.h"
#include "gpio.h"
#include "occs.h"
#include "sci.h"
#include "sys.h"
#include "freemaster.h"

/*
 * Test structure types - demonstrates the "TSA" feature thanks to which the
 * FreeMASTER is able to load a variable and type information directly from
 * the embedded application.
 *
 */

typedef struct {

  UWord16 aa;
  UWord32 bb[2];
  Word16  cc;
  Word32  dd[3];
  UWord8  ee;
  Word8   ff[5];
} INNER_STRUCT;

typedef struct {

  UWord16 a;
  UWord32 b;
  INNER_STRUCT inA[4];
  INNER_STRUCT inB;
} OUTER_STRUCT;

/* Test variables, will be displayed in the FreeMASTER application */
volatile UWord8  var8;
```

```
volatile UWord16 var16;
volatile UWord16 var16inc = 1;
volatile UWord32 var32;
volatile UWord32 var32inc = 100;
volatile UWord8 nAppCmdCounter;

/* Strucutre type information will be available in the FreeMASTER application (TSA) */
OUTER_STRUCT so1, so2;
INNER_STRUCT si1, si2;

/* local prototypes */
void init_pins(void);
void init_uart(void);

////////////////////////////////////////////////////////////////////////

/*
 * With TSA enabled, the user describes the global and static variables using
 * so-called TSA tables. There can be any number of tables defined in
 * the project files. Each table does have the identifier which should be
 * unique across the project.
 *
 * Note that you can declare variables as Read-Only or Read-Write.
 * The FreeMASTER driver denies any write access to the Read-Only variables
 * when TSA_SAFETY is enabled.
 */

FMSTR_TSA_TABLE_BEGIN(first_table)
      FMSTR_TSA_RO_VAR(var8,     FMSTR_TSA_UINT8)
      FMSTR_TSA_RO_VAR(var16,    FMSTR_TSA_UINT16)
      FMSTR_TSA_RO_VAR(var32,    FMSTR_TSA_UINT32)
      FMSTR_TSA_RW_VAR(var16inc, FMSTR_TSA_UINT16)
      FMSTR_TSA_RW_VAR(var32inc, FMSTR_TSA_UINT32)
      FMSTR_TSA_RW_VAR(so1,      FMSTR_TSA_USERTYPE(OUTER_STRUCT))
      FMSTR_TSA_RW_VAR(si1,      FMSTR_TSA_USERTYPE(INNER_STRUCT))

      FMSTR_TSA_STRUCT(OUTER_STRUCT)
      FMSTR_TSA_MEMBER(OUTER_STRUCT, a,   FMSTR_TSA_UINT16)
      FMSTR_TSA_MEMBER(OUTER_STRUCT, b,   FMSTR_TSA_UINT32)
      FMSTR_TSA_MEMBER(OUTER_STRUCT, inA, FMSTR_TSA_USERTYPE(INNER_STRUCT))
      FMSTR_TSA_MEMBER(OUTER_STRUCT, inB, FMSTR_TSA_USERTYPE(INNER_STRUCT))

      FMSTR_TSA_STRUCT(INNER_STRUCT)
      FMSTR_TSA_MEMBER(INNER_STRUCT, aa, FMSTR_TSA_UINT16)
      FMSTR_TSA_MEMBER(INNER_STRUCT, bb, FMSTR_TSA_UINT32)
      FMSTR_TSA_MEMBER(INNER_STRUCT, cc, FMSTR_TSA_SINT16)
      FMSTR_TSA_MEMBER(INNER_STRUCT, dd, FMSTR_TSA_SINT32)
      FMSTR_TSA_MEMBER(INNER_STRUCT, ee, FMSTR_TSA_UINT8)
      FMSTR_TSA_MEMBER(INNER_STRUCT, ff, FMSTR_TSA_SINT8)
FMSTR_TSA_TABLE_END()

/*
 * This is an example of another TSA table. Typically, you put one table
 * to each .c file where your global or static variables are instantiated.
 */

FMSTR_TSA_TABLE_BEGIN(next_table)
      FMSTR_TSA_RO_VAR(so2, FMSTR_TSA_USERTYPE(OUTER_STRUCT))
      FMSTR_TSA_RO_VAR(si2, FMSTR_TSA_USERTYPE(INNER_STRUCT))
FMSTR_TSA_TABLE_END()

////////////////////////////////////////////////////////////////////////

/*
 * This list describes all TSA tables which should be exported to the
 * FreeMASTER application.
```

```
 */

FMSTR_TSA_TABLE_LIST_BEGIN()
       FMSTR_TSA_TABLE(first_table)
       FMSTR_TSA_TABLE(next_table)
FMSTR_TSA_TABLE_LIST_END()

////////////////////////////////////////////////////////////////////////

/*
 * This function is registerred as a application command handler (see
 * main() below. It gets automatically invoked when the PC FreeMASTER
 * application sends appropriate application command.
 *
 */

FMSTR_APPCMD_RESULT myhandler(FMSTR_APPCMD_CODE nAppcmd, FMSTR_APPCMD_PDATA pData,
FMSTR_SIZE nDataLen)
{
       // the return value is used as the application command result code
       return 0x10;
}

////////////////////////////////////////////////////////////////////////

int main(void)
{
       UWord16 nAppCmdCode;

       // initialize SYS and GPIO modules
       init_pins();

       // initialize UART
       init_uart();

       // FreeMASTER initialization
       FMSTR_Init();

       // registerring the App.Command handler
       FMSTR_RegisterAppCmdCall(10, myhandler);

       // initialize & enable interrupts
       ioctl(INTC, INTC_INIT, NULL);
       archEnableInt();

       // other initializations
       var8 = 10;

       // main application loop
       while (1)
       {
              // scope variables
              var16 += var16inc;
              var32 += var32inc;

              // the application commands not registerred with callback handlers
              // can be detected and processed using the API calls below
              // first, check if a new command has been received
              nAppCmdCode = FMSTR_GetAppCmd();

              // when a new command arrives, the nAppCmdCode contains the application
              // command code. In other case, the "NOCMD" special value is returned
              if (nAppCmdCode != FMSTR_APPCMDRESULT_NOCMD)
              {
                     nAppCmdCounter++;
```

```
                    // each command may have deifferent processing and different
                    // result code. The command processing is finished by
                    // calling FMSTR_AppCmdAck() with the result code value
                    switch(nAppCmdCode)
                    {
                          case 1: FMSTR_AppCmdAck(var8); break;
                          case 2: FMSTR_AppCmdAck(~var8); break;
                          default: FMSTR_AppCmdAck(0); break;
                    }
             }

             // This call should be placed in the timer interrupt or anywhere where
             // the recorder sampling should occur.
             FMSTR_Recorder();

             // The FreeMASTER poll call must be called in the main application loop
             // to handle the communication interface and protocol.
             // In LONG_INTR FreeMASTER interrupt mode, all the processing is done
             // during the communication interrupt routine and the FMSTR_Poll() is
             // compiled empty.
             FMSTR_Poll();
       }
}

/*
 *  Initialize SYS and all GPIO modules.
 */

void init_pins(void)
{
       ioctl(SYS, SYS_INIT, NULL);

       // This code is targeted to all 56F8xxx-based evaluation boards.
       // On some processors, some GPIO instances are not implemented
#ifdef GPIO_A
       ioctl(GPIO_A, GPIO_INIT, NULL);
#endif
#ifdef GPIO_B
       ioctl(GPIO_B, GPIO_INIT, NULL);
#endif
#ifdef GPIO_C
       ioctl(GPIO_C, GPIO_INIT, NULL);
#endif
#ifdef GPIO_D
       ioctl(GPIO_D, GPIO_INIT, NULL);
#endif
#ifdef GPIO_E
       ioctl(GPIO_E, GPIO_INIT, NULL);
#endif
#ifdef GPIO_F
       ioctl(GPIO_F, GPIO_INIT, NULL);
#endif
}

/*
 *  Initialize the UART ports
 */

void init_uart(void)
{
       // initialize UART
       ioctl(SCI_0, SCI_INIT, NULL);

       // Initialize SCI_1 as well if such a module exists -
       // this may be useful on boards with SCI_1 RS232 port
```

```
#ifdef SCI_1
      ioctl(SCI_1, SCI_INIT, NULL);
#endif
}
```

The FreeMASTER driver configuration extracted from the appconfig.h file:

```
/*.
    FMSTR Configuration
------------------------------------------
.*/
#define FMSTR_COMM_INTERFACE             1
#define FMSTR_LONG_INTR                  0
#define FMSTR_SHORT_INTR                 1
#define FMSTR_POLL_DRIVEN                0
#define FMSTR_USE_APPCMD                 1
#define FMSTR_USE_SCOPE                  1
#define FMSTR_USE_TSA                    1
#define FMSTR_USE_TSA_SAFETY             1
```

Chapter 7
Graphical Configuration Tool

7.1 Introduction

This section describes the functionality of the Graphical Configuration Tool (GCT) for the 56F8xxx family of Digital Signal Controllers. The Graphical User Interface (GUI), settings, file formats and the import/export rules will be described. The features and functionality of on-chip peripheral modules are described in the **56F8300 Peripheral User Manual** (83xx and 81xx devices) and **56F8000 Peripheral Reference Manual** (80xx devices).

GCT enables to create a static configuration of on-chip peripheral modules in an easy-to-use graphical environment. A graphical representation exists for all control bits or registers of each peripheral module. On the other hand, the status bits or bit-fields and other run-time-only registers are not covered by GCT. Typically, the run-time registers are accessed indirectly by the “ioctl” calls described in previous chapters.

7.1.1 Features

Main GCT features:

- Easy-to-use graphical environment
- Convenient navigation to individual peripheral modules
- Possibility to integrate GCT into CodeWarrior IDE
- Immediate register values view
- Configuration warnings list

Supported peripheral modules:

- Processor block view, package pinout view
- On-chip Clock Synthesis (OCCS)
- System Integration Module (SIM, SYS)
- External Memory Interface (SEMI)
- Computer Operating Properly Watchdog (COP)
- Interrupt controller (INTC)
- Pulse Width Modulation (PWM)

- General Purpose Input/Output (GPIO)
- Analog-to-Digital Converter (ADC)
- Quadrature Decoder (DEC)
- Serial Communication Interface (SCI)
- Serial Peripheral Interface (SPI)
- Inter-integrated Circuit Interface (IIC)
- Quad Timer Module
- FlexCAN Module
- Temperature Sensor

7.1.2 How does it work?

The Graphical Configuration Tool (GCT) is a standard Microsoft Windows-based application used to graphically edit (read and write) project's *appconfig.h* file. The register initialization values edited by graphical controls can be immediately displayed and/or written back to the *appconfig.h* file.

The *appconfig.h* file is included by the application source code, and the register initialization values are used in the "init" functions to physically configure the peripheral module. An initialization function exists for each module and it is typically invoked using the "ioctl" INIT call (e.g. use the SCI1_INIT ioctl command to initialize the SCI module 1).

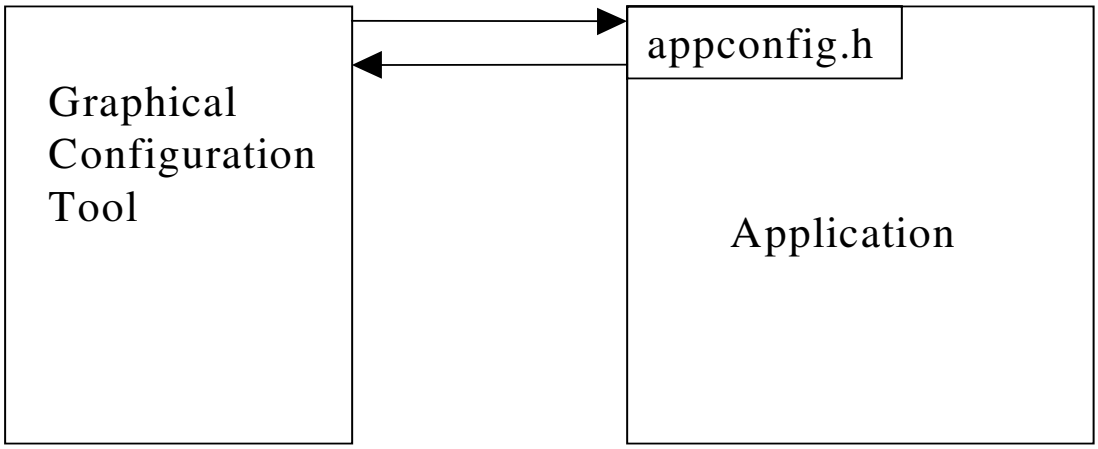


Figure 7-1. GCT Usage

The GCT is able to work as a stand-alone tool, however the integration with the CodeWarrior IDE markedly increases the tool efficiency. The integration steps, as well as the installation, are described in detail in Section 1.2.2.2.

7.2 Program usage

This chapter contains a detailed description of the GCT user interface.

7.2.1 GUI Description

The GUI is organized as a common MS Windows application. The main window displays the following components (see Figure 7-2 below):

- **Peripheral Modules Tree** - The peripheral modules tree enables navigation through the structure of device modules. The user switches between different control pages by clicking on tree items.
- **Information Pane** - The upper left part of main application window displays the type of processor used, processor system clock and peripheral bus clock.
- **Module Settings Pane** - This pane displays the graphical controls which are used to configure the peripheral module.
- **Register Values View** - Displays immediate register values. This bar can be hidden if it is not needed.
- **Warning List** - This side bar displays a list of all configuration conflict warnings.

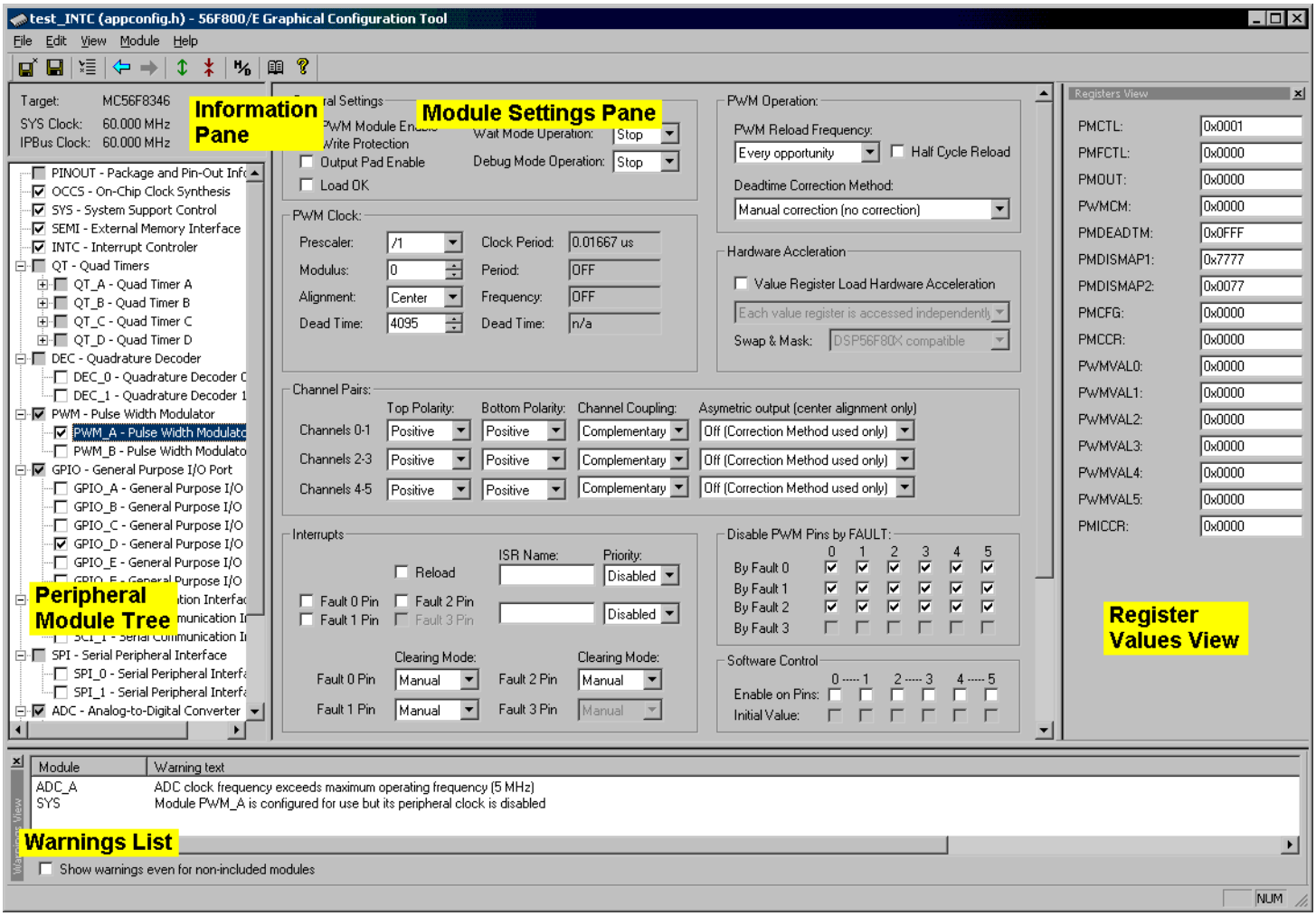


Figure 7-2. GCT Main Window

7.2.1.1 Peripheral Modules Tree

This part of the main application window contains a tree-like list of on-chip peripherals. There is a check-box control at each item, which can be used to enable or disable an *appconfig.h* output for a given module. When the box is unchecked, no module's register values are written to the output file. The GCT warns the user when there are changes made to a module configuration and the configuration is not selected to be saved.

7.2.1.2 Peripheral Module Settings Pane

This is the area where peripheral module control pages are displayed. Default configuration displayed in the GCT control page is a post-reset state of each peripheral module.

The graphical representation of peripheral module settings uses standard MS Windows control elements:

- **Edit Box** as numeric value or string input - numeric values are parsed as decimals by default. To specify hexadecimal number format use the standard "0x" prefix. The **H/D** toolbar button can be used to convert the value just being edited between these two formats.

  When using the Edit Box to specify an interrupt service routine (ISR) name in the GCT, use the valid C-language function name, without the types, parameters or even the brackets.
- **Drop-Down List Box** (Combo Box) - used as mode selectors, typically affecting a multiple-bits fields in a control register.
- **Check Box** - typically representing a single-bit configuration values

7.2.1.3 Pinout Page

The pinout page is a "welcome" page displayed initially when the GCT is started. This page shows a block diagram of the processor device as it is known from the data sheet.

The page content is actively generated by the GCT and displays several useful information about how the device is currently configured. For each peripheral module, the page displays a status icon which informs the user about what package pins are being used by the module and how they are configured.

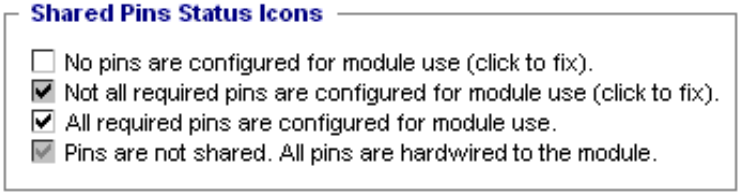

Shared Pins Status Icons
No pins are configured for module use (click to fix).
Not all required pins are configured for module use (click to fix).
All required pins are configured for module use.
Pins are not shared. All pins are hardwired to the module.

Figure 7-3. Pinout Page Status Icons

The page can also be switched to a "package view" where a top-view of the device package is displayed. Each package pin on this view is labeled by a function name assigned by pin multiplexer in a current project. Clicking on the pin label activates the GPIO control page where a pin multiplexer configuration can be changed.

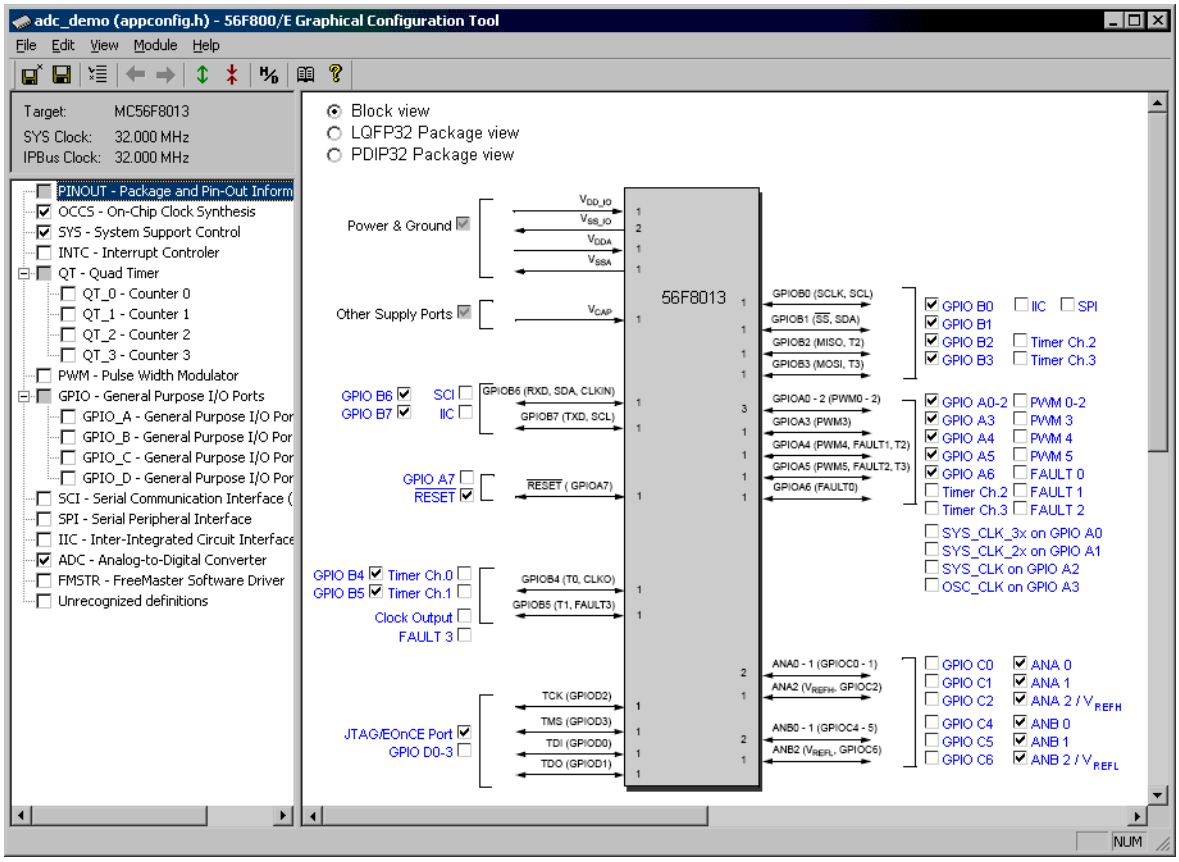


Figure 7-4. Pinout Page

7.2.1.4 Register View

On any of the GCT control pages, a **Register View** bar can be shown to display the immediate register values as they are to be written to the output file. For each module, all registers bound to graphical controls on the page are displayed. When a module configuration is modified, the affected registers values are red-highlighted in the Register View.

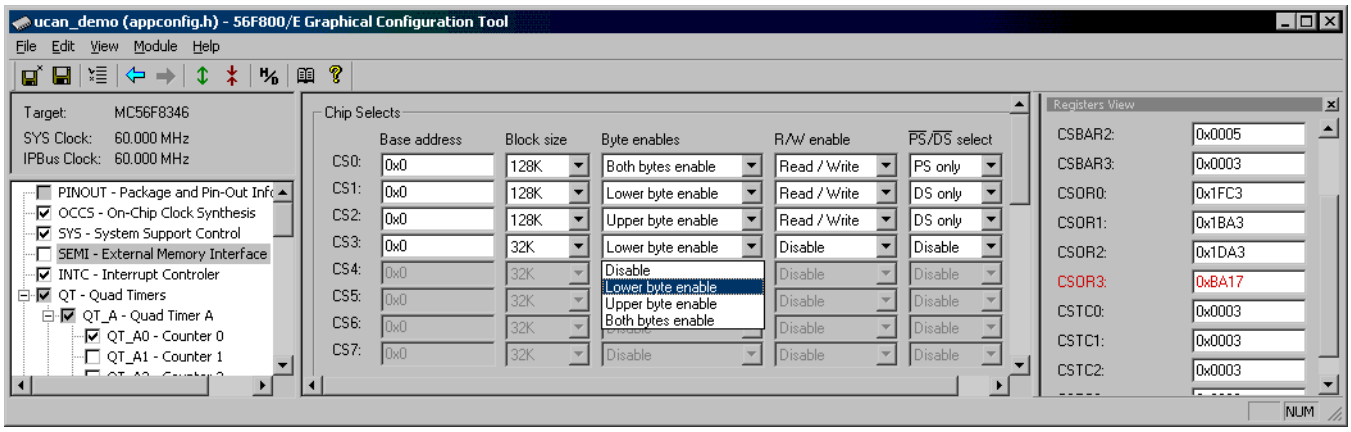

ucan_demo (appconfig.h) - 56F800/E Graphical Configuration Tool
File Edit View Module Help
Target: MC56F8346
SYS Clock: 60.000 MHz
IPBus Clock: 60.000 MHz
PINOUT - Package and Pin-Out Inf
OCCS - On-Chip Clock Synthesis
SYS - System Support Control
SEMI - External Memory Interface
INTC - Interrupt Controler
QT - Quad Timers
QT_A - Quad Timer A
QT_A0 - Counter 0
QT_A1 - Counter 1
Chip Selects
Base address
Block size
Byte enables
R/W enable
PS/DS select
CS0: 0x0 128K Both bytes enable Read / Write PS only
CS1: 0x0 128K Lower byte enable Read / Write DS only
CS2: 0x0 128K Upper byte enable Read / Write DS only
CS3: 0x0 32K Lower byte enable Disable Disable
CS4: 0x0 32K Disable Disable Disable
CS5: 0x0 32K Lower byte enable Disable Disable
CS6: 0x0 32K Upper byte enable Disable Disable
CS7: 0x0 32K Both bytes enable Disable Disable
Registers View
CSBAR2: 0x0005
CSBAR3: 0x0003
CSOR0: 0x1FC3
CSOR1: 0x1BA3
CSOR2: 0x1DA3
CSOR3: 0xBA17
CSTC0: 0x0003
CSTC1: 0x0003
CSTC2: 0x0003
NUM

Figure 7-5. Register View

There is also a possibility to modify the register values directly in the Register View window (press “Enter” to accept a new value), causing the graphical controls to be redrawn accordingly. However, there are often other “run-time” bits in the control registers, which are not supposed to be set during an initialization. Modifying the register values without paying a high attention to each individual bit or bit-field of the register may cause the module settings to be invalid - even if the configuration looks good in the GCT.

The Register View bar can be activated or deactivated by a menu *“View / Register Summary”*.

7.2.1.5 Warnings View

Similarly as the Registers View bar, the **Warnings View** bar can be shown or hidden any time when working with the GCT. Warnings View shows a list of warnings collected from across all control pages in the GCT. By default, the Warnings View displays only the warnings from modules selected to be saved to an output file (those with a “checkmark” sign in peripheral tree). All other warnings can be displayed in the list if required.

A double-click on a warning item in the list activates the control page where the potential conflict exists and a balloon-like hint is shown as a notification.

The Warnings View bar can be activated or deactivated by a menu *“View / Warnings Summary”*.

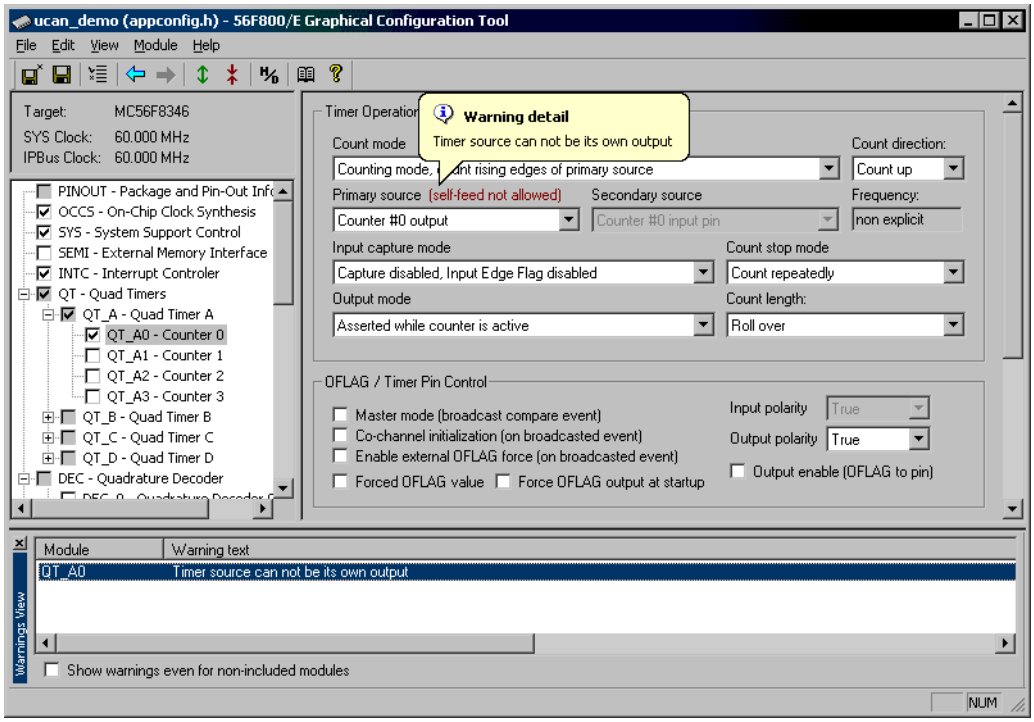


Figure 7-6. Warnings View

7.2.1.6 Options dialog

The Options dialog (Figure 7-7) can be invoked from a *“File / Options...”* menu. It contains the following settings for the overall GCT functionality:

- **Generate detailed comments** check-box enables saving of the “human-readable” commentary describing the configuration of each module.

- **Preserve user comments** check-box, when checked assures the user comments placed after the individual macro values are not lost when re-generating the *"appconfig.h"* file. This option is rarely used as there is typically no need for the user to manually edit any comments in the *"appconfig.h"* file.
- **Generate all register values** check-box enables saving of *all* register values of the selected peripheral modules into the *"appconfig.h"* file. When this option is not enabled (which is the default state), only the registers with non-reset values are saved. Omitting the reset-value registers in the *"appconfig.h"* file typically reduces a size of the module initialization code as those values not defined in the file are not written to the peripheral registers. On the other side, this approach requires **all the modules** being initialized to **be in post-reset state**, otherwise the result may be a most probably a mix of previous registers state and a GCT-defined configuration.

  This is a project-specific option, which is saved to the *"appconfig.h"* file as a macro constant named *APPCFG_DFLTS_OMITTED*. This constant is set to non-zero value when the **Generate all register values** option is turned off**.**
- **Set as default** check-box to save the selected options as default ones for the GCT and the current project.

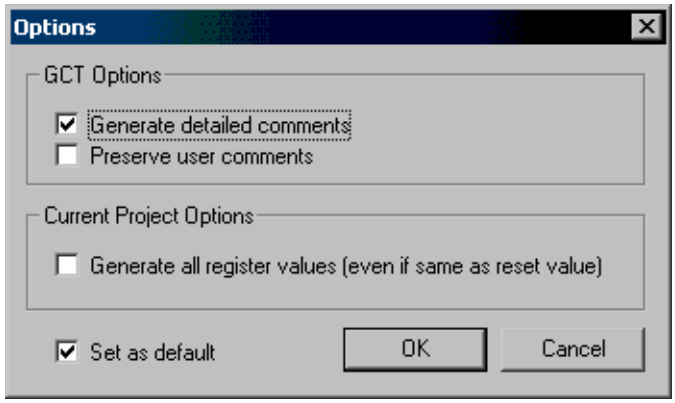


Figure 7-7. Options dialog

7.2.2 Application Configuration File Structure

The application configuration header file is a common ANSI C-language header file, which mostly contains register initialization values, but may contain other application global declarations or macros. The GCT parses and re-generates the register-values part of the file, while the other parts are ignored and left untouched.

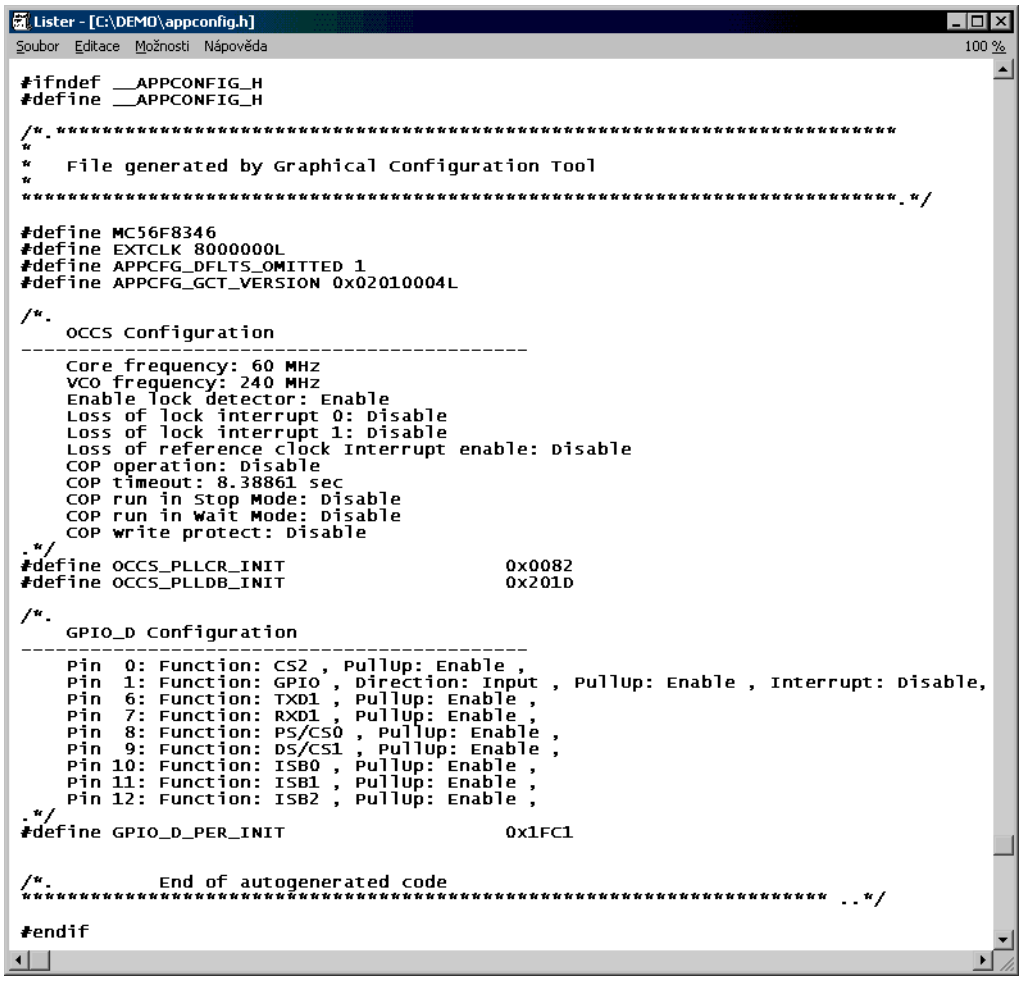
In the Quick_Start projects, the application configuration file is named *"appconfig.h"* and resides in the *"ApplicationConfig"* project folder subdirectory. This file is included by all Quick_Start low-level driver source files where a configuration information is required.

However, the GCT can be used independently on the Quick_Start and its low-level drivers. The GCT is capable of loading and saving the configuration file under any name.

The configuration file can be logically divided into four parts:

- ***Heading*** is an arbitrary text before the first #define statement. Typically, the heading contains the application or user-specific file header or commentary. This part is not modified by the GCT. The first #define is typically a macro used to protect the file from multiple inclusion during compilation (i.e. standard #ifndef/#define sequence).

- ***General GCT Constants*** - in which the processor type and external crystal clock frequency are specified as macros.
- ***Module Configuration Sections*** - a block of configuration macro constants generated for each peripheral module. This part of the file is terminated with special comment ("/* end of auto-generated code ..*/).
- ***File Tail*** - is anything what follows the configuration sections. The GCT keeps this section unchanged so the user may put his own global declarations and statements here. This feature enables a user to use the *appconfig.h* as a central application configuration file included also by his project source files and not only by "Quick_Start" driver files.


Lister - [C:\DEMO\appconfig.h]
Soubor Editace Možnosti Nápověda
100 %
#ifndef __APPCONFIG_H
#define __APPCONFIG_H
/*.***
*
*   File generated by Graphical Configuration Tool
*
***.*/
#define MC56F8346
#define EXTCLK 8000000L
#define APPCFG_DFLTS_OMITTED 1
#define APPCFG_GCT_VERSION 0x02010004L
/*.
    OCCS Configuration

    Core frequency: 60 MHz
    VCO frequency: 240 MHz
    Enable lock detector: Enable
    Loss of lock interrupt 0: Disable
    Loss of lock interrupt 1: Disable
    Loss of reference clock Interrupt enable: Disable
    COP operation: Disable
    COP timeout: 8.38861 sec
    COP run in Stop Mode: Disable
    COP run in Wait Mode: Disable
    COP write protect: Disable
.*/
#define OCCS_PLLCR_INIT 0x0082
#define OCCS_PLLDB_INIT 0x201D
/*.
    GPIO_D Configuration

    Pin 0: Function: CS2 , PullUp: Enable ,
    Pin 1: Function: GPIO , Direction: Input , PullUp: Enable , Interrupt: Disable,
    Pin 6: Function: TXD1 , PullUp: Enable ,
    Pin 7: Function: RXD1 , PullUp: Enable ,
    Pin 8: Function: PS/CS0 , PullUp: Enable ,
    Pin 9: Function: DS/CS1 , PullUp: Enable ,
    Pin 10: Function: ISB0 , PullUp: Enable ,
    Pin 11: Function: ISB1 , PullUp: Enable ,
    Pin 12: Function: ISB2 , PullUp: Enable ,
.*/
#define GPIO_D_PER_INIT 0x1FC1
/*.   End of autogenerated code
*** ..*/
#endif

Figure 7-8. The *appconfig.h* File

Chapter 8
License

8.1 Software License Agreement

IMPORTANT. Read the following Freescale Semiconductor Software License Agreement ("Agreement") completely.

FREESCALE SEMICONDUCTOR SOFTWARE LICENSE AGREEMENT

[SOFTWARE FOR: DSP56800E_Quick_Start Tool]

This is a legal agreement between you (either as an individual or as an authorized representative of your employer) and Freescale Semiconductor, Inc. ("Freescale"). It concerns your rights to use this file and any accompanying written materials produced by Freescale (the "Software"). In consideration for Freescale allowing you to access the Software, you are agreeing to be bound by the terms of this Agreement. If you do not agree to all of the terms of this Agreement, do not install or download the Software. If you change your mind later, stop using the Software and delete all copies of the Software in your possession or control. Any copies of the Software that you have already distributed, where permitted, and do not destroy will continue to be governed by this Agreement. Your prior use will also continue to be governed by this Agreement. Please note that 3rd party products, including but not limited to software ("3rd Party Products"), may be distributed in conjunction with the Software. This Agreement does not apply to those 3rd Party Products, which will be subject to their own licensing terms.

LICENSE GRANTS. Your license to the Software and applicable restrictions vary depending on the nature of the Software provided. Review the following grants carefully to ensure your compliance.

IF SOFTWARE PROVIDED IN SOURCE FORM. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale or other development, prototype, or production platform utilizing at least one 56800/E processor from Freescale ("Exclusive Use"), (2) to reproduce the Software as necessary to accomplish the Exclusive Use, (3) to prepare derivative works of the Software as necessary to accomplish the Exclusive Use, (4) to distribute the Software and derivative works

thereof in object (machine-readable) form only as integrated with a development platform from Freescale or other development, prototype, or production platform utilizing at least one 56800/E processor from Freescale, and (5) to sublicense to others the right to use the distributed Software. You must prohibit your sublicensees from translating, reverse engineering, decompiling, or disassembling the Software except to the extent applicable law specifically prohibits such restriction. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

IF SOFTWARE PROVIDED IN OBJECT FORM ONLY. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale or other development, prototype, or production platform utilizing at least one 56800/E processor from Freescale ("Exclusive Use"), (2) to reproduce the Software as necessary to accomplish the Exclusive Use, (3) to distribute the Software only as integrated with a development platform from Freescale or other development, prototype, or production platform utilizing at least one 56800/E processor from Freescale, and (4) to sublicense to others the right to use the distributed Software.

The Software is provided to you only in object (machine-readable) form. You may exercise the rights above only with respect to such object form. You may not translate, reverse engineer, decompile, or disassemble the Software except to the extent applicable law specifically prohibits such restriction. In addition, you must prohibit your sublicensees from doing the same. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

FOR TOOLS. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale ("Exclusive Use"), and (2) to reproduce the Software. The Software is provided to you only in object (machine-readable) form. You may not distribute or sublicense the Software to others. You may exercise the rights above only with respect to such object form. You may not translate, reverse engineer, decompile, or disassemble the Software except to the extent applicable law specifically prohibits such restriction. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

COPYRIGHT. The Software is licensed to you, not sold. Freescale owns the Software, and United States copyright laws and international treaty provisions protect the Software. Therefore, you must treat the Software like any other copyrighted material (e.g., a book or musical recording). You may not use or copy the Software for any other purpose than what is described in this Agreement. Except as expressly provided herein, Freescale does not grant to you any express or implied rights under any Freescale or third party patents, copyrights, trademarks, or trade secrets. Additionally, you must reproduce and apply any copyright or other proprietary rights notices included on or embedded in the Software to any copies or derivative works made thereof, in whole or in part, if any.

SUPPORT. Freescale is NOT obligated to provide any support, upgrades or new releases of the Software. If you wish, you may contact Freescale and report problems and provide suggestions regarding the Software. Freescale has no obligation whatsoever to respond in any way to such a problem report or suggestion. Freescale may make changes to the Software at any time, without any obligation to notify or provide updated versions of the Software to you.

LIMITED WARRANTY ON MEDIA. Freescale warrants that the media on which the Software is recorded will be free from defects in materials and workmanship under normal use for a period of 90 days from the date of purchase as evidenced by a copy of the receipt.

Freescale's entire liability and your exclusive remedy under this warranty will be replacement of the defective media returned to Freescale with a copy of the receipt. Freescale will have no responsibility to replace any media damaged by accident, abuse or misapplication. This warranty extends only to you and may be invoked only by you for your customers. Freescale will not accept warranty returns from your customers.

NO ADDITIONAL WARRANTY. EXCEPT FOR THE LIMITED WARRANTY ON MEDIA PROVIDED ABOVE, THE SOFTWARE AND 3RD PARTY PRODUCTS, IF ANY, ARE PROVIDED "AS IS". YOUR USE OF THE SOFTWARE OR 3RD PARTY PRODUCTS IS AT YOUR SOLE RISK. SHOULD THE SOFTWARE OR ANY 3RD PARTY PRODUCT PROVE DEFECTIVE, YOU (AND NOT FREESCALE OR ANY FREESCALE REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FREESCALE EXPRESSLY DISCLAIMS ALL WARRANTIES WITH RESPECT TO THE SOFTWARE AND 3RD PARTY PRODUCTS, WHETHER SUCH WARRANTIES ARE EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. YOU EXPRESSLY ASSUME ALL LIABILITIES AND RISKS, FOR ANYONE'S USE OR OPERATION OF ANY APPLICATION PROGRAMS YOU MAY CREATE WITH THE SOFTWARE.

INDEMNITY. Freescale will defend, at its expense, any suits asserted against you based upon a claim that the Software as provided by Freescale infringes a U.S. patent or copyright or misappropriates a trade secret, and pay costs and damages finally awarded based upon such suit, if you: (1) promptly notify Freescale in writing as soon as reasonably practicable after you first become aware of the claim of infringement or misappropriation, but in no event later than 15 days of the date on which you first received notice of the claim; and (2) at Freescale's request and expense, give Freescale sole control of the suit and all requested assistance for defense of the suit. Freescale will not be liable for any settlement made without its written consent. If the use or sale of any Software component program licensed under this Agreement is enjoined as a result of such suit, Freescale at its option and at no expense to you, will: (1) obtain for you the right to use such program consistent with the license granted in this Agreement for the affected program; (2) substitute an equivalent program and extend this indemnity thereto; or (3) accept the return of the program and refund the portion of the license fee for such component program less reasonable charge for prior use. If an infringement or misappropriation claim related to the Software is alleged prior to completion of delivery, Freescale has the right to decline to make further shipments notwithstanding any other provision of this Agreement. This indemnity does not extend to any suit based upon any infringement or alleged infringement arising from any program furnished by Freescale that is: (1) altered in any way by you or any third party if the alleged infringement would not have occurred but for such alteration; (2) combined with any other

products or elements not furnished by Freescale if the alleged infringement would not have occurred but for such combination; (3) designed or manufactured in accordance with your designs, specifications or instructions if the alleged infringement would not have occurred but for such designs, specifications or instructions; or (4) designed or manufactured in compliance with standards issued by any public or private standards body if the alleged infringement would not have occurred but for compliance with such standards. In no event will Freescale indemnify you or be liable in any way for royalties payable based on a per use basis, or any royalty basis other than a reasonable royalty based upon revenue derived by Freescale from your license of the Software.

THE INDEMNITY PROVIDED IN THIS SECTION IS THE SOLE, EXCLUSIVE, AND ENTIRE LIABILITY OF FREESCALE AND THE REMEDIES PROVIDED IN THIS SECTION SHALL BE YOUR EXCLUSIVE REMEDIES AGAINST FREESCALE FOR PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION AND IS PROVIDED IN LIEU OF ALL WARRANTIES, EXPRESS, IMPLIED OR STATUTORY IN REGARD THERETO, INCLUDING, WITHOUT LIMITATION, THE WARRANTY AGAINST INFRINGEMENT SPECIFIED IN THE UNIFORM COMMERCIAL CODE.

LIMITATION OF LIABILITY. IN NO EVENT WILL FREESCALE BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY INCIDENTAL, SPECIAL, INDIRECT, CONSEQUENTIAL OR PUNITIVE DAMAGES, INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR ANY LOSS OF USE, LOSS OF TIME, INCONVENIENCE, COMMERCIAL LOSS, OR LOST PROFITS, SAVINGS, OR REVENUES TO THE FULL EXTENT SUCH MAY BE DISCLAIMED BY LAW.

COMPLIANCE WITH LAWS; EXPORT RESTRICTIONS. You must use the Software in accordance with all applicable U.S. laws, regulations and statutes. You agree that neither you nor your licensees (if any) intend to or will, directly or indirectly, export or transmit the Software to any country in violation of U.S. export restrictions.

GOVERNMENT USE. Use of the Software and any corresponding documentation, if any, is provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(l) and (2) of the Commercial Computer Software--Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Freescale Semiconductor, Inc., 6501 William Cannon Drive West, Austin, TX, 78735.

HIGH RISK ACTIVITIES. You acknowledge that the Software is not fault tolerant and is not designed, manufactured or intended by Freescale for incorporation into products intended for use or resale in on-line control equipment in hazardous, dangerous to life or potentially life-threatening environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines or weapons systems.

CHOICE OF LAW; VENUE; LIMITATIONS. You agree that the statutes and laws of the United States and the State of Texas, USA, without regard to conflicts of laws principles, will apply to all matters relating to this Agreement or the Software, and you agree that any litigation will be subject to the exclusive jurisdiction of the state or federal courts in Texas, USA. You agree that regardless of any statute or law to the contrary, any claim or cause of action arising out of or related to this Agreement or the Software must be filed within one (1) year after such claim or cause of action arose or be forever barred.

PRODUCT LABELING. You are not authorized to use any Freescale trademarks, brand names, or logos.

ENTIRE AGREEMENT. This Agreement constitutes the entire agreement between you and Freescale regarding the subject matter of this Agreement, and supersedes all prior communications, negotiations, understandings, agreements or representations, either written or oral, if any. This Agreement may only be amended in written form, executed by you and Freescale.

SEVERABILITY. If any provision of this Agreement is held for any reason to be invalid or unenforceable, then the remaining provisions of this Agreement will be unimpaired and, unless a modification or replacement of the invalid or unenforceable provision is further held to deprive you or Freescale of a material benefit, in which case the Agreement will immediately terminate, the invalid or unenforceable provision will be replaced with a provision that is valid and enforceable and that comes closest to the intention underlying the invalid or unenforceable provision.

NO WAIVER. The waiver by Freescale of any breach of any provision of this Agreement will not operate or be construed as a waiver of any other or a subsequent breach of the same or a different provision.